Язык программирования AWL:
предварительное описание

Версия: ***Март'2014***

Введение

Язык программирования AWL — это интерпретируемый язык (или *язык сценариев*), предназначенный для решения разнообразных задач обработки числовых и символьных данных. AWL во многом основан на идеях других языков (таких, как C, Java, LISP, Perl, JavaScript и других) — но заметно отличается от всех перечисленных, имея ряд собственных интересных особенностей. Одна из перспективных целей AWL — быть основой альтернативной технологии создания гипертекстовых документов, предлагающей единую интегрированную среду для описания структурного документа и средств динамического управления его контентом.

Здесь описано ядро языка — та функциональная основа, на которой могут быть реализованы прикладные AWL-среды.

Рабочая среда AWL

Язык AWL реализован и функционирует как *интерпретатор*.
С точки зрения операционной системы, AWL-интерпретатор — это прикладная программа (в Windows, обычно, **awl.exe**; в UNIX-системах —просто **awl**). Интерпретатор может работать либо в в *интерактивном* режиме, либо в *пакетном*. Общий синтаксис обращения к интерпретатору таков:

```
awl.exe [Module.awl Arg1 Arg2 ... ArgN]
```

Первый аргумент в списке (*Module.awl*) — имя файла выполняемого AWL-модуля, следующие — список *аргументов*, которые ему передаются. Работа интерпретатора в пакетном режиме состоит в том, что он выполняет команды из *Module.awl*, после чего завершается.
Если аргументы в командной строке отсутствуют, интерпретатор будет работать в *интерактивном* режиме: последовательно читать вводимые пользователем AWL-выражения, распознавать и выполнять их. Далее, диалог пользователя с интерпретатором будет показываться в таком виде:

```
ввод пользователя ->
  -> вывод интерпретатора
```

По умолчанию, при интерактивной работе включен режим *трассировки*: это означает, что после каждого шага интерпретатор выводит выражение и результат его вычисления (в виде: *выражение => результат*). Простейший диалог пользователя с AWL-интерпретатором может выглядеть, например, так:

```
2*2 + 3*3;
  add:(mul:(2, 2), mul:(3, 3)) => 13;

sqr(5) + sqr(7);
  add:(sqr:5, sqr:7) => 4.8818193;

'Hello' +$ ', ' +$ 'world!';
  s_cat:(s_cat:("Hello", ", "), "world!") => "Hello, world!";
```

Заметьте, что каждое вводимое пользователем выражение должно завершаться точкой с запятой. (Сложное выражение может быть разбито на любое количество строк — конец строки в AWL всегда равносилен пробельному символу.) В качестве приглашения для ввода AWL-интерпретатор выводит текущий номер выполняемой команды (в квадратных скобках).
По умолчанию, как сами выражения, так и результаты выводятся во *внутреннем* (приведенном) формате, который (как правило) отличается от формата ввода. (Внутренний формат подробно описан в дальнейших главах книги.)
Диалог с интерпретатором может продолжаться сколь угодно долго. Способ его завершения в общем случае зависит от системы. В Windows необходимо ввести строку, состоящую из одного символа **^Z** (**Ctrl + Z**), в UNIX-системах — **^D** (**Ctrl + D**, нажимать на «ввод» необязательно).

По завершении работы, интерпретатор обычно выводит краткую финальную статистику (затраченное время, использование памяти и т.п.)

Программный код и данные

Язык AWL поддерживает разнообразные типы данных. К простейшим (т. н. *скалярным*) относятся *числа* и *строки*, к более сложным — *списки* и другие *агрегаты данных*. Принципиальной особенностью AWL (сближающей его с такими языками, как LISP и Scheme) является то, что все выражения в AWL также являются структурами данных. В отличие от многих языков, AWL не знает четкого разграничения между кодом и данными: фактически, сама AWL-программа также является структурой данных (обычно довольно сложной). Хотя это требуется не очень часто, программный код можно даже конструировать непосредственно в процессе выполнения.
Любое выражение языка может иметь как *значение* (результат его вычисления), так и *побочный эффект* вычисления. Первый аспект характеризует его как компонент данных, второй — как компонент исполняемого программного кода. Заметим, что иногда вычисление выражения осуществляется ради получения результата, иногда — ради его побочного эффекта, а иногда — для обеих целей.
Мы начнем знакомство с языком с рассмотрения простейших структур данных.

Скалярные типы

Скаляры — это простейшие элементы данных, не обладающие внутренней структурой. Все скаляры также неизменяемы (*иммутабельны*). Их семантика выполнения всегда тривиальна: скаляр всегда вычисляется сам в себя. В свою очередь, скаляры подразделяются на *числовые* и *строковые*.

Числовые данные

Числовые скаляры представляют собой просто числа — *целые* или с *плавающей точкой*. Строго говоря, эти два подтипа числового типа *не идентичны*: внутреннее представление для целых и вещественных чисел различается. Обычно, это не играет существенной роли, так как язык часто обеспечивает автоматическое преобразование (*приведение*) числовых значений. Но в ряде случаев различие в семантике целых и вещественных типов бывает важным.
Литеральная запись для числового скаляра имеет традиционный для большинства языков программирования вид. Вот примеры целых скаляров:

```
1
928
3026
```

Целочисленный литерал может задаваться не только в десятичном виде:

\o*OOOO*	*Восьмеричное целое*: последовательность восьмеричных цифр *OOOO*
\x*XXXX* **\h***XXXX*	*Шестнадцатеричное целое*: последовательность шестнадцатеричных цифр *XXXX*
\b*BBBB*	*Двоичное целое*: последовательность двоичных цифр *BBBB*

Внутреннее представление целых чисел — 32-битовое со знаком. Допустимый диапазон целых значений — от -2 147 483 648 до +2 147 483 647 включительно.

Более широкий диапазон имеют *вещественные числа*. Вот примеры вещественных литералов:

```
3.14159
0.0243e7
145.3497e-12
```

Представление вещественных чисел соответствует стандарту *IEEE-754* двойной точности (64 бита). Это обеспечивает большой диапазон представления значений, вплоть до +/- 1,79769313486232e308 (интервал строго симметричен относительно нуля).
Формат вывода для числовых данных обычно мало отличается от формата ввода. Вещественные числа могут выводиться или с фиксированной точкой (*DDD.DD*), или в нормализованном экспоненциальном формате (*DDD.DDeNN*), если интерпретатору это покажется удобным. Перед отрицательным значением выводится знак.
Специальными вещественными значениями также являются *положительная/отрицательная бесконечность* (+**Inf**/-**Inf**) и *арифметическая неопределенность*, или «не число» (**NaN**).

Строковые данные

Строковые скаляры — это символьные строки неограниченной (практически) длины. Количество символов в строке называется ее *длиной* (очевидно, что она всегда неотрицательна, и пустая строка имеет длину 0). В AWL (в отличие от C), строка может содержать любые символы (включая нулевой — т. е. имеющий код 0). Диапазон допустимых символов определяется т. н. *типом строки*:

- **тип 0**: строка может содержать только ASCII-символы (коды 0..255)
- **тип 1**: строка может содержать только символы Unicode-16 (коды 0..65535)
- **тип 2**: строка может содержать полный набор символов Unicode (без ограничений)

В большинстве случаев, преобразование между строками этих типов также осуществляется *автоматически* (хотя есть операции и для явного преобразования). Тем не менее, помнить про тип строки желательно.
При вводе, строковый литерал может ограничиваться либо одинарными, либо двойными кавычками. Если для ограничения литерала применяются одинарные кавычки, внутри могут содержаться двойные (и наоборот).
Символ '\' в строке является специальным: он дает возможность включать в нее ряд управляющих символов:

\a	[код 7]	*звуковой сигнал*
\n	[код 10]	*новая строка*
\r	[код 13]	*возврат каретки*
\t	[код 9]	*горизонтальная табуляция*
\b	[код 8]	*шаг назад*
\e	[код 27]	*escape*
\f	[код 12]	*form feed*
\v	[код 11]	*вертикальная табуляция*
\x*XX* **\h***XX*	Символ с *шестнадцатеричным* кодом *XX* (до 2 шестнадцатеричных цифр)	

\o*OOO*	Символ с *восьмеричным* кодом *OOO* (до 3 восьмеричных цифр)
\d*DDD*	Символ с *десятичным* кодом *DDD* (до 3 десятичных цифр)
\c*C*	*Управляющий символ* (соответствующий **Ctrl**+*C* при вводе с клавиатуры)
\[*NAME***]**	Символ с *символическим именем NAME* (полный список здесь не приводится)

Кроме того, символ обратной косой черты отменяет специальную интерпретацию кавычек (одинарных и двойных), и самого себя.
По умолчанию, строковый литерал имеет тип 0 (т. е. может включать в себя только символы ASCII). Если литерал должен включать в себя символы Unicode, ему должен предшествовать префикс **\u** или **\U**.
Вот примеры строковых литералов:

```
''        `(пустая строка)`
"A"
'ABCD'
"1x, 2y, 3z"
'< --- >'
"Hello, world!"
"Это строка #1,
 это строка #2
 и это строка #3."
```

Из последнего примера видно, что строковый литерал может занимать *несколько строк* исходного кода (в этом случае символы конца строк включаются в него *буквально*).

Формат, в котором интерпретатор выводит строки — также не содержит особых сюрпризов. В качестве ограничителей всегда используются двойные кавычки («»), а многие управляющие символы могут кодироваться символами из вышеприведенной таблицы. Все символы, выходящие за пределы набора ASCII, выводятся в 16-ричном виде: “**x**XXXX”.

Списки

Одной из самых важных структур данных языка является *список*. Список — это последовательность из более чем одного *элемента*. Элементами списка могут быть скаляры, другие списки, и практически любые другие иные структуры данных. Таким образом, списки могут быть вложенными (причем глубина их вложенности, как и длина, не ограниченна).
В исходном коде список в общем случае задается *конструктором списка*. Простейшая форма конструктора списка — это перечисление элементов списка, разделенных *запятыми*, и заключенных в *круглые скобки*. Вот несколько примеров задания списков:

```
(567, "aa bb cc dd", 432, '9 8 7', 123.321)
((10, 'Aa'), (25, 'Bb'), (31, 'Cc'), 0)
('1', ('2.1', '2.2'), (('3.1.1', '3.1.2', '3.1.3'), ('3.2.1',
'3.2.2', '3.2.3')), 0)
```

Приведенные списки являются *замкнутыми*, и имеют (соответственно) 5, 4 и 4 элемента.

Заметим, что в последнем случае — если б отсутствовал конечный элемент 0, список неявно принял бы такой вид:

```
('1', ('2.1', '2.2'), ('3.1.1', '3.1.2', '3.1.3'), '3.2.1',
'3.2.2', '3.2.3')
```

Дело в том, что когда последний элемент списка сам является списком — он неявным образом «продолжает» исходный список. Таким образом, все нижеследующие списки на деле совершенно идентичны:

```
(10, (20, (30, (40, 50))))
(10, (20, (30, 40, 50)))
(10, (20, 30, 40, 50))
(10, 20, 30, 40, 50)
```

Фактически, это — четыре разных способа задать один и тот же список из пяти чисел: (10, 20, 30, 40, 50). Но что делать, если последним элементом списка — также должен быть список? Для этого существует специальная разновидность списков — *открытые* списки:

```
(10, (20, 30, 40, 50), )
(10, 20, (30, 40, 50), )
(10, 20, 30, (40, 50), )
(10, 20, 30, 40, 50, )
```

Все перечисленные списки являются *открытыми*, различаются, и имеют (соответственно) 2, 3, 4 и 5 элементов (причем во всех примерах, кроме последнего — последнем элементом также является список). Впрочем (как мы увидим далее) фактическим последним элементом этих списков является *пустое значение* () — однако (по соглашению) большинство списковых операций *не рассматривают* его как полноценный элемент, и игнорируют. (У открытых списков есть и некоторые другие преимущества перед замкнутыми — но они будут подробнее объяснены позднее.)

Синтаксически конструктор списка имеет самый низкий приоритет — т.е. элементами списка могут быть практически любые выражения языка. Но в тех случаях, когда все элементы списка являются *синтаксически замкнутыми* выражениями — допускается более компактная запись для списков. В ней в качестве ограничителей используются *квадратные скобки*, а элементы списка разделяются *пробельными символами* (или их последовательностями):

```
[567 "aa bb cc dd" 432 '9 8 7' 123.321]
```

Синтаксически замкнутыми выражениями являются все литералы, переменные, другие списки (в любой из двух перечисленных форм) и *блоки*. Заметим, что эта «замкнутость» является чисто синтаксическим понятием: любое выражение легко сделать «замкнутым», просто заключив его в круглые скобки. Таким образом, следующие два списка тоже совершенно эквивалентны:

```
(2*a - 10, 7*b + 5, 10*c - 2)
[(2*a - 10) (7*b + 5) (10*c - 2)]
```

Но в этом случае запись с круглыми скобками выглядит естественнее (и, разумеется, короче). По умолчанию, список в квадратных скобках также считается *закрытым*. Если надо задать открытый список, имеется специальный синтаксис: перед закрывающей список скобкой

ставится *двоеточие*. Следующие два списка также равносильны:

```
('aa', 'bbb', 'cccc', 'ddddd', )
['aa' 'bbb' 'cccc' 'ddddd' :]
```

Поскольку списки играют в языке исключительно важную роль, они могут быть записаны и другими способами (но с ними мы познакомимся чуть позднее).
Как уже говорилось, у каждого выражения в AWL имеется не только результат, но и определенная *семантика выполнения*. Для списков эта семантика проста и самоочевидна: выполнение списка состоит в последовательном выполнении всех его элементов, от первого до последнего. (Понятно, что если элементами списка также являются списки, эта логика применяется к ним рекурсивно.) Результатом выполнения/вычисления списка становится список из результатов всех его элементов (имеющий такую же структуру). В частности, поскольку все скаляры самовычисляемы, то результатом вычисления списка, содержащего только скаляры — является совершенно идентичный список (другими словами, список из самовычисляемых элементов также самовычисляем).
Независимо от того, каким способом список был задан (или получен) — он выводится интерпретатором в стандартной форме: в круглых скобках и с элементами, разделенными запятыми. (Если список является открытым, запятая следует и после последнего элемента.)

Пустое значение

В языке имеется *пустое* значение, обозначающее отсутствие значения как такового. Оно во многом аналогично атому **nil** в LISP/Scheme. В исходном коде его можно явно задать в виде () или []. Во многих случаях его также можно рассматривать как пустой список (что технически не корректно, т. к. настоящим списком оно не является).
В AWL пустое значение — это способ указать на то, что явное значение *отсутствует*. Всякое вычисляемое выражение в AWL возвращает некое значение (по определению) — но, когда это значение не нужно или не существенно, им обычно является (). Аналогично, () является исходным значением для всех *переменных* (как и прочих *мутабельных* объектов языка). (В AWL никогда не бывает «мусора» (т. е. неопределенного значения), как в C и некоторых других процедурных языках. Если переменная (или что-нибудь еще) не получила явного значения — этим значением обычно является ().)
Наконец (как и в других случаях) у пустого значения имеется тривиальная *семантика выполнения*: как легко догадаться, при его выполнении ничего не происходит. Поэтому, оно дополнительно также играет роль *пустой инструкции* в большинстве языков.
Пустое значение () не следует путать с другими «пустыми» значениями языка: т. е. с целым нулем (0), вещественным нулем (0.0), арифметической неопределенностью (**NaN**) и с пустой строкой («»). В отличие от пустого значения — все перечисленные являются полноценными *скалярами*.
Как и следует ожидать, интерпретатор всегда выводит пустое значение () также в виде ().

Идентификаторы

Идентификаторы — это символические имена. Все идентификаторы AWL обозначают либо *переменные*, либо *функторы*. Синтаксис идентификаторов в AWL довольно традиционен: идентификатор состоит из букв латинского алфавита (заглавных или строчных), десятичных цифр и знаков подчеркивания (но при этом он не должен начинаться с цифры!) Очень важно, что при этом заглавные и строчные буквы *различаются*. Вот примеры разрешенных идентификаторов:

```
A
xyz
average_value
Link123
ParagraphCounter
CP_1252
```

Идентификаторы всегда интерпретируются в контексте определенного *пространства имен*. При этом имена *переменных* и имена *функторов — всегда* относятся к двум *разным* пространствам имен (которые никогда не пересекаются). Самым внешним пространством имен является *глобальное*. К нему принадлежат, например, глобальные переменные текущего *модуля*, или же текущей (активной) *сессии* интерпретатора.
Во многих языках в качестве идентификаторов не допускаются *зарезервированные* (ключевые) слова. В AWL нет такого ограничения, т.к. ключевые слова в языке в принципе *отсутствуют*. Даже имена всех встроенных функторов являются всего лишь стандартными идентификаторами, которые даже можно переопределить (хотя делать это без серьезной необходимости не рекомендуется).
Идентификатор из единственного подчеркивания (“_”) также вполне допустим. Однако, по соглашению его рекомендуется использовать лишь в специальных случаях. Называть так обычные переменные не рекомендуется.

Пробелы и комментарии

В промежутках между литералами, идентификаторами и символами синтаксиса допускаются произвольные *пробельные символы*: собственно *пробелы*, *табуляции*, *концы строк* и некоторые другие управляющие символы. В ряде случаев пробелы *необходимы* — например, там, где они являются единственными разделителями (скажем, между элементами списка, заключенного в квадратные скобки). Любая последовательность пробельных символов равносильна одному пробелу (за исключением строковых литералов, конечно).
Там, где язык допускает пробел — допускается и *комментарий*. Комментарий — это произвольная непустая последовательность символов, заключенная в *обратные кавычки* (и не содержащая таких кавычек сама).

```
` Это - простой комментарий `
```

Комментарий также вполне может занимать несколько строк программного кода подряд. Содержимое комментариев полностью игнорируется, и никак не влияет на выполнение программы.

Функторы и термы

Все функциональные объекты в AWL называются *функторами*. Язык содержит большое количество *встроенных* функторов (их имена находятся в глобальном пространстве имен, и доступны в любой точке программы). Помимо встроенных (предопределенных) функторов — пользователь может определять свои собственные.
Вызов функторов обеспечивают *термы*. Собственно, терм — это совокупность *имени функтора* (встроенного или пользовательского) и *аргумента* (которым может быть список). Как и все прочее, терм также является полноценной структурой данных. Однако, у них есть нетривиальная семантика исполнения: данному функтору передается данный аргумент, и происходит *вычисление*, возвращающее *результат* (возможно, пустой). Для многих функторов определен и *побочный эффект*.
Вот несколько примеров термов:

```
add(A, B)
mul(sub(A, 2), add(C, 3))
date(2006, 1, 15)
title("Месячный отчет: ", bold(Year), "-", bold(Month))
```

На самом деле, выполнение AWL-программы в основном состоит из вычисления или выполнения разнообразных функторов. Встроенными функторами являются, например, все операции и управляющие структуры. По этой причине, для многих часто употребляемых встроенных функторов имеется и *альтернативный* (более традиционный) синтаксис, который мы рассмотрим ниже. Однако, независимо от внешнего синтаксиса, внутренним результатом всегда будет терм, реализующий вызов некоторого функтора с операндом(ами).
Когда функтор ожидает более чем один операнд — ему необходимо передать *список* (из двух, трех и более) *операндов*. Количество требуемых операндов принято называть *арностью* функтора. Так функторы, требующие один аргумент, называются *унарными*; два аргумента — *бинарными*; три аргумента — *тернарными*. Если функтор вообще не требует аргументов (такие тоже имеются) — он считается *нульарным*. Конечно, не все встроенные функторы имеют жестко заданную арность: многие могут работать со списками аргументов переменной длины.
При вызове унарного функтора, имеющего аргументом *скалярный литерал* (т. е. число или строку), скобки вокруг последнего могут быть опущены:

```
sin 1;
```

```
sin:1 => 0.84147098;
```

Кроме того, предусмотрен еще один способ записи терма: вместо **func**(arg) всегда можно написать **func**^arg. В случае, когда операндом является список — это явное синтаксическое излишество. Но когда единственным операндом терма является другой терм, это бывает удобно. Например, нижеследующие выражения эквиваленты:

```
l_tail (l_tail (l_tail (LongList)))
l_tail^l_tail^l_tail^LongList
```

Независимо от того, каким именно образом терм был задан в исходном коде — интерпретатор всегда выводит его в следующем виде:

```
functor:argument
```

(Заметим, что при вводе — такой синтаксис как раз не допустим!)

Блоки

Рассмотрим последнюю из встроенных конструкций AWL — *блок*. Подобно спискам и термам, блок — это одновременно и структура данных, и структура управления (хотя употребляется в основном для последней цели).
Синтаксически *блок* — это произвольная последовательность произвольных выражений (элементов), заключенная в фигурные скобки, и *разделенная* точками с запятой. Блоки (как и списки) являются синтаксически замкнутыми. Вычисление (или выполнение) блока состоит в последовательном вычислении/выполнении всех его элементов. При этом значение *последнего* выполненного элемента — возвращается в качестве результата самого блока.
Например, в приведенном случае результатом блока будет значение `c =/: 2`

```
{ a = b + c; b =*: 2; c =/: 2 }     ` пример блока `
```

Таким образом, блок — это основное средство для организации (линейной) *последовательности* вычислений. Собственно, аналогичного эффекта можно добиться и с помощью списка (как уже сказано, при вычислении списка также последовательно вычисляются все его элементы). Но имеется и существенная разница: вычисление списка возвращает результаты вычисления *всех его элементов*, а вычисление блока — *только последнего элемента* (остальные результаты игнорируются). Если же значение последнего элемента блока также несущественно — он может быть пустым (в этом случае, вычисление блока возвращает значение ()).
Таким образом, важно понимать разницу между:

```
{ S1; S2; S3 … Sn }
```

и

```
{ S1; S2; S3 … Sn; }
```

В обоих случаях семантика выполнения *одинакова* (последовательно выполняются все элементы блока, от *S1* до *Sn*), но возвращаемый результат *различается* (в первом случае — им будет результат *Sn*, а во втором — всегда (), т. е. пустота). Отсюда ясно, что если значение последнего элемента блока *требуется* в дальнейших вычислениях, ставить в конце точку с запятой *ни в коем случае нельзя* (т. к. тогда последнее значение просто потеряется). Еще раз напомним, что она является *разделителем* элементов блока (как в Паскале или Модуле), но не их *завершителем* (как в Java или C)! Наоборот: если возвращаемое блоком значение принципиально несущественно (а важно только выполнение всех его элементов), то поставить точку с запятой в конце — *хороший тон*, т. к. явно подчеркивает, что блок не возвращает значения.
Бессмысленными (но вполне законными!) конструкциями является блок из одного единственного элемента:

```
{ S }
```

(его выполнение просто сводится к выполнению *S*) и *пустой* блок:

```
{}
```

(как и при выполнении (), при выполнении пустого блока ничего не происходит).
Формат вывода блоков интерпретатором мало отличается от формата ввода: также применяются фигурные скобки, и точки с запятой в качестве разделителей. Но (для улучшения читаемости) каждый элемент блока выводится с новой строки, а перед ними добавляются отступы, пропорциональные глубине вложенности блоков.

Программа и ее выполнение

Таким образом, практически все важнейшие структурные единицы программ и данных (скаляры, списки, термы и блоки) мы уже рассмотрели. Осталось сказать несколько слов о выполнении программы в целом.
Обычно AWL-программа представляет собой единственный файл (*модуль*) исходного кода. Он, так же как и блок, представляет собой последовательность исполняемых инструкций, разделенных точками с запятыми. Все инструкции — это либо произвольные *выражения*, либо *определения* (для функторов, классов и пр.). Конечно, любое выражение/инструкция

может возвращать некоторое значение (которое просто игнорируется).
Завершив выполнение модуля, интерпретатор выводит некую краткую финальную статистику.
Пусть у нас есть файл (скажем, **test.awl**) с таким исходным кодом :

```
a = 12; b = 23;
a*2 + b*3;
(a - b) * (a + b)
```

результат его обработки интерпретатором будет выглядеть примерно так:

```
<test> :: {
set:(a, 12) => 12;
set:(b, 23) => 23;
add:(mul:(a, 2), mul:(b, 3)) => 93;
mul:(sub:(a, b), add:(a, b)) => -385;
} :: #4 / [4]
```

Здесь <test> — имя этого модуля (оно обычно определяется именем файла), [4] — общее число выполненных инструкций, и #4 — общее число строк в исходном модуле.

Скалярные операции

Простейшими элементами данных в AWL являются *скаляры*. Соответственно, простейшие операции — *скалярные*, для которых и операнд(ы), и результат имеют некий скалярный тип. Все эти операции реализованы как *встроенные функторы*, имеющие явные имена. Однако, использующие эти операции термы часто могут быть записаны и в неявном («операторном») виде: как префиксные, постфиксные или инфиксные *операции*. С точки зрения семантики обе формы записи совершенно равносильны, какую из них предпочесть — вопрос удобства. Далее в таблицах для каждой операции приведена как *нормальная* (т. е. «функциональная»), так и ее «операторная» форма (если отдельный синтаксис для нее предусмотрен языком).
При разборе выражений, записанных в операторной форме — учитываются *приоритеты* операций. Все операции разбиты на следующие основные приоритетные группы (перечисленные в порядке уменьшения приоритета):

- *унарные* (префиксные/постфиксные);
- *мультипликативные* (группа умножения/деления и сдвигов);
- *аддитивные* (группа сложения/вычитания);
- *максимум* и *минимум*;
- *компаративные* (группа сравнений);
- *битовые* и *логические*;
- *управляющие* (условные и циклические);
- *присваивания* и операции ввода/вывода.

Бинарные операции имеют левую ассоциативность (если явно не указано обратное).

Арифметические скалярные операции

Для всех этих операций как операнд(ы), так и возвращаемый результат являются *числами* (целыми или вещественными). Здесь (и далее) для задания их операндов будут использоваться следующие *мета-переменные*:

- `A, B, C`: числовые операнды (целые или вещественные);
- `I, J, K`: целые операнды (или приводимые к целым);
- `X, Y, Z`: вещественные операнды (или приводимые к вещественным).

Если реальный тип операнда не совпадает с требуемым — выполняются *приведения* (о них подробно рассказано ниже).

Унарные арифметические операции:		
Синтаксис	***Нормальная форма***	***Семантика***
`- A`	**`neg`**`(A)`	*Арифметическое отрицание A*
`+ A`	**`abs`**`(A)`	*Абсолютная величина (модуль) A*
`<?> A`	**`sgn`**`(A)`	*Знак A:* -1, 0 *или* 1

Бинарные мультипликативные операции:		
Синтаксис	***Нормальная форма***	***Семантика***
`A * B`	**`mul`**`(A,B)`	*Произведение A и B*
`X / Y`	**`div`**`(X,Y)`	*Частное X и Y*
`I % J`	**`idiv`**`(I,J)`	*Целочисленное частное I и J*

Бинарные мультипликативные операции:		
`I %% J`	**irem**`(I,J)`	*Целочисленный остаток I и J*
`I << J`	**shl**`(I,J)`	*Арифметический сдвиг I (влево на J битов)*
`I >> J`	**shr**`(I,J)`	*Арифметический сдвиг I (вправо на J битов)*

Обратите внимание на синтаксис операций целочисленного деления и остатка! При выполнении целого деления — происходит округление результата к нулю (как и в C). Для операций целого деления и остатка всегда справедливо следующее тождество: `I == (I%J) * J + (I%%J)`. Целочисленное деление на 0 выдает ошибку.
Для операций сдвигов — второй операнд может быть отрицательным. Для них справедливы такие тождества: `I << J == I >> (-J)` и `I >> J == I << (-J)`. Сдвиг влево/вправо на 0 битов ничего не меняет: `I << 0 == I >> 0 == I`.

Бинарные аддитивные операции:		
Синтаксис	***Нормальная форма***	***Семантика***
`A + B`	**add**`(A,B)`	*Сумма A и B*
`A - B`	**sub**`(A,B)`	*Разность A и B*

Менее стандартные операции — это числовые *максимум* и *минимум*:

Бинарные операции максимума и минимума:		
Синтаксис	***Нормальная форма***	***Семантика***
`A ?< B`	**min**`(A,B)`	*Минимум (меньшее из A и B)*
`A ?> B`	**max**`(A,B)`	*Максимум (большее из A и B)*

Арифметические операции сравнения

Для этих операций операнды также должны принадлежать к числовому типу. Результатом всегда является целое значение 0 (если сравнение *ложно*), или 1 (если оно *истинно*). (Такие операции называются *предикатами*.)

Бинарные операции арифметического сравнения (компаративные):		
Синтаксис	***Нормальная форма***	***Семантика***
`A < B`	**lt**`(A,B)`	*A меньше B?*
`A > B`	**gt**`(A,B)`	*A больше B?*
`A <= B` `A ~> B`	**le**`(A,B)`	*A меньше или равно B?*
`A >= B` `A ~< B`	**ge**`(A,B)`	*A больше или равно B?*

Бинарные операции арифметического сравнения (компаративные):		
`A == B`	**`eq`**`(A,B)`	*A равно B?*
`A ~= B` `A <> B`	**`ne`**`(A,B)`	*A не равно B?*
`A <?> B`	**`cmp`**`(A,B)`	*Результат сравнения A и B*

Последняя операция (**cmp**) возвращает не логическое значение (0: истина / 1: ложь), а знак: -1 (если `A < B`); 1 (если `A > B`) или 0 (если `A == B`). Она полезна в основном для операций сортировки.
Следующие функторы (нульарные) дают некоторые целые константы:

Наибольшее/наименьшее целое значение	
Нормальная форма	***Семантика***
`max_int``()`	*Наибольшее возможное целое*
`min_int``()`	*Наименьшее возможное целое*

Ни одно из целых не может быть меньше, чем **min_int**(), или больше, чем **max_int**().

Битовые операции

Все эти операции также требуют операнд(ы) целого типа. Они рассматриваются, как строки битов, и возвращается целый результат, основанный на побитовой арифметике.

Унарные битовые операции:		
Синтаксис	***Нормальная форма***	***Семантика***
`~ I`	**`not`**`(I)`	*Битовая инверсия для I (NOT)*

Бинарные битовые операции:		
Синтаксис	***Нормальная форма***	***Семантика***
`I & J`	**`and`**`(I,J)`	*Битовое AND (“и”) для I и J*
`I | J`	**`or`**`(I,J)`	*Битовое OR (“или”) для I и J*
`I ~ J`	**`xor`**`(I,J)`	*Битовое XOR (“исключающее или”) для I и J*

(*Внимание*: в отличие от C и Java, символом для операции XOR является '~', а не '^'!)
Из операторной формы любое выражение автоматически приводится к нормальной форме: например, выражение `A*B + C/D` равносильно **`add`**(**`mul`**`(A,B)`, **`div`**`(C,D)`).
Однако (помимо перечисленных), имеется еще ряд встроенных (унарных и бинарных) операций, не имеющих специальной операторной формы (и доступных только в нормальной форме):

Математические функции:	
Нормальная форма	***Семантика***

Математические функции:	
floor(X)	*Округление X вниз до ближайшего целого*
ceil(X)	*Округление X вверх до ближайшего целого*
sqr(X)	*Квадратный корень из X*
exp(X)	*Экспонента от X*
log(X)	*Натуральный логарифм от X*
exp_by(X,Y)	*X в степени Y*
log_by(X,Y)	*Логарифм Y по основанию X*
sin(X)	*Синус X*
cos(X)	*Косинус X*
tan(X)	*Тангенс X*
asin(X)	*Арксинус X*
acos(X)	*Арккосинус X*
atan(X)	*Арктангенс X*
sinh(X)	*Гиперболический синус X*
cosh(X)	*Гиперболический косинус X*
tanh(X)	*Гиперболический тангенс X*
rad(X,Y)	*Радиус для вектора (X, Y):* *(**sqr** (X*X + Y*Y))*
ang(X,Y)	*Угол для вектора (X, Y)*
pi(X)	*Число π, умноженное на X*

Если операнд функции выходит за пределы допустимого диапазона (например, отрицателен для **sqr** или **log**) — обычно результатом является «не число» **NaN**. Аргумент для всех тригонометрических функций (и результат для всех обратных тригонометрических, и для **ang**) выражен в радианах.

Наконец, некоторые стандартные числовые константы также всегда доступны в виде нульарных (т. е. не требующих аргументов) функций:

Функторы / числовые константы:	
Нормальная форма	***Семантика***
inf_pos()	*Положительная бесконечность*
inf_neg()	*Отрицательная бесконечность*
nan()	*Не число (арифметическая неопределенность)*

Строковые скалярные операции

Рассмотрим теперь операции над *строками*. Как и числовые операции, они реализованы как встроенные функторы. Их имена обычно начинаются префиксом **s_**, а их операторные символы содержат суффикс '$' (что позволяет отличать их от числовых операций). У них та

же система приоритетов, что и у арифметических операций. (Далее S, T, U — произвольные операнды строкового типа.)

Основные строковые операции:			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`#$ S`	**`s_len`**`(S)`	унарная	*Длина строки S (целое число)*
`+$ S`	**`s_type`**`(S)`	унарная	*Тип строки S (целое 0, 1 или 2)*
`~$ S`	**`s_rev`**`(S)`	унарная	*Реверсия (переворот) строки S*
`S +$ T`	**`s_cat`**`(S,T)`	аддитивная	*Конкатенация (сцепление) строк S и T*
`S *$ N`	**`s_rep`**`(S,N)`	мультипликативная	*Репликация (повторение) строки S N раз подряд*

Вот примеры использования этих операций.

```
"Black" +$ " and" +$ " white";
```

```
s_cat:(s_cat:("Black", " and"), " white") => "Black and
white";
```

```
"No! " *$ 3;
```

```
s_rep:("No! ", 3) => "No! No! No! ";
```

```
~$ "abracadabra";
```

```
s_rev:"abracadabra" => "arbadacarba";
```

Конкатенация пустой строки (справа или слева) с произвольной строкой возвращает эту строку. Репликация пустой строки (любое количество раз), или ее реверсия — также возвращают пустую строку.

Строковые сравнения, максимум и минимум

Для строк также определены свои операции сравнения. Сравнение строк имеет *лексикографическую* или *словарную* семантику: т. е. строки считаются упорядоченными по словарному принципу, в соответствии с кодами символов. Как и их арифметические аналоги, операции сравнения строк возвращают значение 0 (если сравнение ложно) или 1 (если оно истинно). (Единственное исключение — операция **s_cmp**, возвращающая значение -1, 0 или 1, в зависимости от словарной упорядоченности своих операндов.)

Бинарные операции сравнения строк (компаративные):		
Синтаксис	***Нормальная форма***	***Семантика***
`S <$ T`	**`s_lt`**`(S,T)`	*S раньше T словарно?*
`S >$ T`	**`s_gt`**`(S,T)`	*S позже T словарно?*

Бинарные операции сравнения строк (компаративные):		
`S <=$ T` `S ~>$ T`	**`s_le`**`(S,T)`	*S раньше (или совпадает с) T словарно?*
`S >=$ T` `S ~< T`	**`s_ge`**`(S,T)`	*S позже (или совпадает с) T словарно?*
`S ==$ T`	**`s_eq`**`(S,T)`	*S совпадает с T словарно?*
`S <>$ T` `S ~= T`	**`s_ne`**`(S,T)`	*S не совпадает с T словарно?*
`S <?>$ T`	**`s_cmp`**`(S,T)`	*Результат словарного сравнения S и T*: -1 (если *S* <$ *T*), 1 (если *S* >$ *T*) или 0

Кроме того, как и для чисел, для строк определены операции *словарного* максимума и минимума:

Бинарные операции строкового максимума/минимума:		
Синтаксис	***Нормальная форма***	***Семантика***
`S ?<$ T`	**`s_min`**`(S,T)`	*Меньшая (словарно) из строк S и T*
`S ?>$ T`	**`s_max`**`(S,T)`	*Большая (словарно) из строк S и T*

Сравнения строк на максимум/минимум подчиняются тем же правилам, что и прочие.

Приведения и проверки типов

Как легко видеть, большинство встроенных функторов-операций имеет жестко определенную семантику, включающую определенные типы своих операндов. Поэтому в AWL невозможны ситуации (нередкие, например, в Python и JavaScript), когда семантика операции — числовая она или строковая — зависит от типов операндов периода выполнения. Например, операция «+» всегда возвращает число (сумму своих операндов), а операция «+$» возвращает строку (конкатенацию своих операндов).

Однако, ситуация, когда тип операнда не совпадает с ожидаемым — не считается ошибкой: в этом случае происходит *неявное приведение* значения к требуемому операцией типу. Приведения работают для всех скалярных типов, в соответствии со следующими правилами:

- целые числа могут преобразовываться в действительные, а действительные — в целые. В последнем случае происходит простое отбрасывание дробной части числа (т.е. округление к нулю). Если число выходит за пределы диапазона, допустимого для целых значений — возвращается, соответственно, **min_int**() или **max_int**().
- числовые значения могут преобразовываться в строки. Применяются абсолютно те же правила, что и при выводе числа интерпретатором. (Формат вывода примерно соответствует С-формату «%8g».)
- строковые значения могут преобразовываться в числа. Точнее говоря — преобразуется та начальная часть строки, которая может быть интерпретирована как целое или действительное число (соответственно, и результат будет целым или действительным). Начальные пробельные символы строки игнорируются, начальный знак («+» или «-») допускается. Символы подчеркивания («_») в числовой константе допустимы (и игнорируются). Если распознать число не удалось — возвращается значение 0.

В этом примере, хотя оба операнда — строки, но сама операция сложения является числовой (поэтому результат — также число):

```
'222' + '113';
  add:("222", "113") => 335;
```

А вот и противоположный случай: числовые операнды для строковой операции (конкатенации):

```
156 +$ 208;
  s_cat:(156, 208) => "156208";
```

Нередко встречаются случаи, когда нужны *явные преобразования типов*. Их обеспечивают следующие (унарные) функторы.

Явные приведения типов	
Нормальная форма	***Семантика***
int(Q)	*Преобразовать скаляр Q в целое число*
float(Q)	*Преобразовать скаляр Q в действительное число*
num(Q)	*Преобразовать скаляр Q в число (целое или действительное)*
string(Q)	*Преобразовать скаляр Q в строку*

Явные преобразования осуществляются в соответствии с теми же правилами, что и неявные. Наконец, имеется ряд функторов-*предикатов*, которые позволяют непосредственно проверить, к какому именно скалярному типу принадлежит их операнд:

Предикаты проверки типа	
Нормальная форма	***Семантика***
is_int(Q)	*Скаляр Q — целое число?*
is_float(Q)	*Скаляр Q — действительное число?*
is_num(Q)	*Скаляр Q — число (целое или действительное)?*
is_string(Q)	*Скаляр Q — строка?*

Как и все предикаты — они возвращают 0 при ложности проверки, и 1 при истинности.

Отрезок строки

Очень часто некая операция должна быть выполнена не над всей строкой, а над каким-нибудь ее непрерывным фрагментом. Его можно получить таким образом:

Отрезок строки		
Синтаксис	***Нормальная форма***	***Семантика***
`S $[R]`	**`s_slice`**`(R,S)`	*Отрезок строки S,* *заданный диапазоном индексов R*

Диапазон (*range*) — это важное понятие, которое будет встречаться еще нередко. Для его задания есть специальный синтаксис:

Синтаксис	***Нормальная форма***	***Семантика***
`From .. To`	`((From), (To))`	*Задает диапазон целых значений от (включая) From до (исключая) To.*

Как легко видеть, фактически диапазон — это просто список из двух целых, определяющих его верхнюю и нижнюю границы. (Что ни в коем случае не надо путать со списком, включающем все числа в перечисленном диапазоне!) Заметим, что нижняя граница диапазона (*From*) всегда включается в него, но верхняя граница (*To*) — не включается! Поэтому, диапазон *From .. To* содержит *To - From* значений (если *From >= To*, диапазон является *пустым*, т. е. не содержит ни одного значения). Например: в диапазон 10 .. 15 входят числа (10, 11, 12, 13, 14). Когда в качестве диапазона задано единственное число N — это считается сокращением для 0..N (т. е. диапазон содержит N чисел от 0 до N-1).
Вернемся к операции **s_slice**: она возвращает фрагмент строки *S*, ограниченный диапазоном индексов *R* (отсчет символов строки всегда ведется с 0). Например:

```
"Hello, world" $[5];
  s_slice:(5, "Hello, world") => "Hello";

"Hello, world" $[7..12];
  s_slice:((7, 12), "Hello, world") => "world";
```

Очевидно, что диапазон 0..#$S (или просто #$S) совпадает со строкой *S*. Если же начало и/или конец диапазона выходят за эти пределы, это не считается ошибкой: в этом случае результат *дополняется* (в начале и/или в конце) требуемым количеством *пробелов*. Таким образом, строка-результат всегда имеет ровно столько символов, сколько требует диапазон. (Если диапазон пуст — результатом операции, естественно, будет пустая строка.)

Поиск подстроки в строке

Разумеется, в AWL имеются операции, позволяющие отыскать подстроку в строке.

Строковые операции поиска		
Синтаксис	***Нормальная форма***	***Семантика***
`S >>$ T`	**`s_findfirst`**`(S,T)`	*Найти первое вхождение подстроки T в строке S*
`S <<$ T`	**`s_findlast`**`(S,T)`	*Найти последнее вхождение подстроки T в строке S*

В операторной форме приоритет этих операций такой же, что и у мультипликативных. Операция **s_findfirst** ищет *первое* вхождение подстроки *T* в строке *S* (вперед с начала); **s_findlast** ищет *последнее* вхождение (назад с конца). Если подстрока найдена, возвращается позиция ее *первого* символа в строке *S* (отсчет символов ведется с 0). Если она не найдена, возвращается -1. (Пустая строка всегда успешно "находится" в начале/конце строки.)

```
"underground" >>$ 'und';
  s_findfirst:("underground", "und") => 0;

"underground" <<$ 'und';
  s_findlast:("underground", "und") => 8;
```

Общий префикс/суффикс строк

В языке имеются операции, позволяющие легко найти общий *начальный* или *конечный* фрагмент двух строк:

Строковые операции префикса/суффикса	
Нормальная форма	***Семантика***
s_common_head(S,T)	*Найти общий префикс строк S и T*
s_common_tail(S,T)	*Найти общий суффикс строк S и T*

Операция **s_common_head** возвращает длиннейший *общий начальный* фрагмент (*«префикс»*) строк *S* и *T*, а **s_common_tail** возвращает длиннейший *общий конечный* фрагмент (*«суффикс»*) строк *S* и *T*. (Обе операции *коммутативны*: результат не зависит от порядка операндов.) Если общий префикс и/или суффикс у строк отсутствует — результатом соответствующей операции будет пустая строка.

```
s_common_head ("aquarium", "aqualung");
  s_common_head:("aquarium", "aqualung") => "aqua";

s_common_tail ("aquarium", "terrarium");
  s_common_tail:("aquarium", "terrarium") => "arium";
```

Применение скалярных операций к спискам

Хотя синтаксическая форма операций обычно компактнее и понятнее, знать их нормальную форму может быть полезно по ряду причин. Например, для всех скалярных операций операндом может быть список из нескольких значений.

Применение любой скалярной бинарной операции к списку, содержащему несколько элементов — возвращает своим результатом список, в котором *первые два элемента* заменены на *результат выполнения* соответствующей операции над ними (при этом "хвост" списка остается неизменным). Более формально, результатом **bin_op**(E1, E2, E3, …, En) будет (**bin_op**(E1, E2), E3, …, En) (где **bin_op** — имя функтора любой бинарной операции).

Например:

```
a = (11, 12, 13, 14);
add(a);
```

```
add:a => (23, 13, 14);
```

```
add^add(a);
```

```
add:add:a => (36, 14);
```

```
add^add^add(a);
```

```
add:add:add:a => 50;
```

```
mul(a);
```

```
mul:a => (132, 13, 14);
```

```
mul^mul(a);
```

```
mul:mul:a => (1716, 14);
```

Аналогично, применение любой унарной операции к списку, содержащему более одного элемента, возвращает в качестве результата список, в котором *первый элемент* заменен на *результат* этой операции над ним (при этом “хвост” списка остается неизменным). Более формально, результатом **un_op**(E1, E2, …, En) будет (**un_op**(E1), E2, …, En) (где **un_op** — функтор любой унарной операции).
Например (для списка из предыдущего примера):

```
neg(a);
```

```
neg:a => (-11, 12, 13, 14);
```

Таким образом, все эти операции рассматривают список-операнд как *стек* (с вершиной в начале!), из которого извлекается операнд (или операнды), и в который помещается результат. (Даже стандартное поведение этих операций может рассматриваться как частный случай спискового, когда список операндов «урезан» до необходимых одного или двух.)
Если необходимо, к элементам списка также автоматически применяются все скалярные преобразования типов:

```
ll = (11, 12, 13, 14);
s_cat(ll);
```

```
s_cat:ll => ("1112", 13, 14);
```

```
s_cat^s_cat(ll);
```

```
s_cat:s_cat:ll => ("111213", 14);
```

```
s_cat^s_cat^s_cat(ll);
```

```
s_cat:s_cat:s_cat:ll => "11121314";
```

Редукция

Как мы видим, когда к списку применяется унарная операция — она фактически действует

только на первый элемент списка (а когда бинарная — только на пару первых элементов). При этом «хвост» списка остается нетронутым.

Но что делать, если нужно применить бинарную операцию ко *всем элементам* списка по порядку? Для этого есть специальная мета-операция: *редукция* («мета-», т. к. ее операндом фактически является другая операция). Синтаксис редукции таков: [=] **bin_op** *List* (где **bin_op —** символ соответствующей бинарной операции), интерпретатором редукция неявно транслируется в свой функциональный эквивалент: **reduce**(**f_bin_op**(List)) (где **f_bin_op** — функтор, соответствующий **bin_op**). Вот пример (сумма всех элементов списка a из предыдущей главы):

```
[=] + a;

reduce:add:a => 50;
```

Редукция работает так: заданная операция выполняется над первыми двумя элементами списка, затем над результатом и следующим элементом — и так далее, пока элементы в списке не кончатся. Понятно, что редукцию имеет смысл применять лишь к бинарным операциям. Когда в списке только два элемента, применение редукции ничем не отличается от применения операции непосредственно к ним. Если редукция применяется к скаляру (или пустому значению) — она просто возвращает свой операнд.

Редукция в принципе применима ко всем скалярным бинарным операциям — но наиболее полезны следующие ее варианты (которые можно считать идиомами):

- `[=] + L` сумма всех элементов в *L*
- `[=] * L` произведение всех элементов в *L*
- `[=] ?> L` максимум из всех элементов в *L*
- `[=] ?< L` минимум из всех элементов в *L*
- `[=] +$ L` конкатенация всех элементов в *L* (как строк)

Как обычно, если для операции требуются скалярные приведения, они выполняются: при редукции числовых операций строки преобразуются в числа, а при редукции строковых — числа в строки.

Переменные и присваивания

Как и в большинстве языков программирования (за исключением разве что «чистых» функциональных, типа Haskell) — в AWL присутствуют *переменные*.
Простейший вид переменных AWL — *глобальные* (доступные во всем текущем модуле, или до завершения сессии интерпретатора). Они обычно не требуют предварительного описания (или объявления), и создаются *автоматически*: первое упоминание переменной с заданным именем вызывает ее к жизни. AWL — *бестиповой язык*, и переменным также не приписаны жестко заданные типы: любая переменная может иметь произвольное значение произвольного типа (и даже менять его в процессе выполнения программы).
Пока переменной не присвоено значение — она содержит (). Основной способ изменения значения переменных — *присваивание.* В AWL (как и в «C-подобных» языках) оно является *операцией* (и, как и прочие операции, реализована *встроенным функтором*). Необычно то, что в языке имеется две разные операции присваивания: *вычисляющее* (**set**) и *не вычисляющее* (**let**).

Операции присваивания:		
Синтаксис	***Нормальная форма***	***Семантика***
`V = W`	**`set`**`(V, W)`	*Присваивание:* *присвоить V вычисленное значение W*

(О «ленивом» присваивании **let** мы подробнее поговорим позже — в главе о ленивых вычислениях.)
Операция присваивания является *мутатором* — то есть функтором, приводящим к изменению данных. (Есть и другие мутаторы, с которыми мы познакомимся чуть позднее.)

Простое присваивание

Простейшая операция присваивания записывается как V = W (в функциональной форме, **set**(V, W)). Ее выполнение состоит в том, что значение операнда *W* вычисляется, и его результат присваивается операнду *V*. Операция требует, чтобы операнд *V* был *мутабельным* (т.е. доступным для изменения). Переменная — это простейший (но не единственный!) вид мутабельного выражения в языке. Вот несколько примеров присваивания значений разным переменным:

```
a = 12*3;
b = 'Hello, ' +$ 'world!';
c = (2*2, 3*3, 4*4);
d = (('x' *$ 3, 'y' *$ 2), 'z' *$ 5)
```

В результате четыре переменные получат следующие значения:

```
a; b; c; d;
```

```
a => 36;
b => "Hello, world!";
c => (4, 9, 16);
d => (("xxx", "yy"), "zzzzz");
```

Как видим, присваиваться могут не только скаляры, но и списки. Интересно то, что первым

операндом присваивания также может быть список: список является мутабельным, если все его элементы также мутабельны. Когда первым и вторым операндом являются списки одинаковой длины — присваивание происходит *поэлементно*:

```
[a b c] = (10.5, 12.9, 14.3);
[d e] = ('Yes', 'No')

a; b; c; d; e;
```

```
a => 10.5;
b => 12.9;
c => 14.3;
d => "Yes";
e => "No";
```

При этом, если какой-нибудь из элементов списка *W* сам является списком, его структура сохраняется при присваивании. Длины левого и правого списка могут не совпадать. Если левый список *короче* правого — его последний элемент получает своим значением все элементы правого списка, оставшиеся не присвоенными. (Таким образом, значения в присваиваемом списке никогда не "пропадают" бесследно. Для Perl-программистов такая семантика присваивания списков хорошо знакома.)

```
[a b c] = [10 22 35 67];

a; b; c;
```

```
a => 10;
b => 22;
c => (35, 67);
```

Если же левый список *длиннее* правого — то всем переменным, которым "не хватит значений", будет присвоено ().

```
[a b c] = ['ABC' 'DEF'];

a; b; c;
```

```
a => "ABC";
b => "DEF";
c => ();
```

Заметим, что, как следствие этого, чтобы *очистить* переменную *V* (т. е. присвоить ей ()), достаточно написать **set** (V,).
Важно отметить и то, что при присваивании списков — все присваивания их элементов выполняются *одновременно*. (Фактически, все значения элементов правого списка вычисляются *до* их присваивания левому.) Например, вот допустимый способ поменять местами значения переменных *x* и *y* (который нетрудно обобщить и на большее количество переменных).

```
[x y] = [y x];
```

(Однако, для быстрого обмена местами двух значений удобнее использовать **swap**, описанный ниже).
Наконец, операция присваивания возвращает значение — то самое, которое было присвоено ее левому операнду. Его можно использовать в дальнейших вычислениях. (При этом обычно надо помнить о скобках, т. к. присваивания имеют более низкий приоритет, чем большинство операций.)

```
X = (Y = 15) + (Z = 25);

(X, Y, Z);
```

```
(X, Y, Z) => (40, 15, 25);
```

Списковое присваивание может быть эффективнее, чем последовательность из скалярных присваиваний. (Фактически, как легко видеть — скалярное присваивание тоже можно рассматривать как частный случай спискового.)

Совмещенное присваивание

Во многих языках программирования (особенно C-подобных) есть возможность совмещения присваивания с выполнением многих бинарных операций. Такая возможность предусмотрена и в AWL (хотя синтаксис используется несколько другой).
Для произвольной бинарной операции **bin_op** вместо `V = V bin_op W` можно написать просто `V = bin_op : W`. Очевидно, что правый операнд *V* должен быть мутабельным, и, кроме того, он должен быть непустым (содержать скаляр). Помимо более компактной записи, *V* вычисляется только один раз (если *V* — переменная, это ничего не меняет, но для более сложных мутабельных выражений это может дать выигрыш в эффективности). Во внутренней форме — это выражение преобразуется в терм **comb**(**f_bin_op**(V, W)).
Бинарной операцией **bin_op** (или функтором **f_bin_op**) могут быть:

- аддитивные: `+` (**add**), `-` (**sub**)
- мультипликативные: `*` (**mul**), `/` (**div**), `%` (**idiv**), `%%` (**irem**)
- максимум/минимум: `?<` (**min**), `?>` (**max**)
- битовые и сдвиги: `<<` (**shl**), `>>` (**shr**), `&` (**and**), `|` (**or**), `~` (**xor**)
- строковые: `+$` (**s_cat**), `*$` (**s_rep**), `?<$` (**s_min**), `?>$` (**s_max**)

Примеры:

```
`(прибавить elem к sum_all)`
sum_all =+: elem;

`(умножить elem на prod_all)`
product_all =*: elem;
`(добавить elem в конец строки join_all с разделителем)`
join_all =+$: elem +$ ", ";
```

Возможно и совмещение присваивания и с *унарными* операциями. Для произвольной унарной операции **un_op** вместо `V = un_op V` можно написать: `V =: un_op`. Операнд *V* также должен быть мутабельным и непустым (и вычисляется только один раз). Как **un_op** допустимы следующие унарные операции:

`+` (**abs**), `-` (**neg**), `~` (**not**).

Во внутренней форме, `V =: un_op` преобразуется в **comb**(**f_un_op**(V)). Например:

```
val =:-   ` (инвертировать знак переменной val) `
```

Как и редукция, совмещенное присваивание является т.н. *мета-операцией* (реализованной через обращение к встроенному функтору **comb**, выполняющему необходимые действия над своими операндами).

Инкремент и декремент

Увеличение и *уменьшение* числового значения на единицу требуется настолько часто, что для него (как и в С-подобных языках) предусмотрены специальные унарные операции. Операции *инкремента* и *декремента* имеются как в префиксной, так и в постфиксной форме.

Инкремент, декремент и очистка		
Синтаксис	***Нормальная форма***	***Семантика***
`++ A`	**inc**(A)	*Инкремент A (префиксный вариант)*
`-- A`	**dec**(A)	*Декремент A (префиксный вариант)*
`A ++`	**inc_p**(A)	*Инкремент A (постфиксный вариант)*
`A --`	**dec_p**(A)	*Декремент A (постфиксный вариант)*
	clr(A)	*Очистить (присвоить значение 0) A*

Все эти операции — также мутаторы (т.е. требуют, чтобы операнд *A* был мутабелен, и имел значение числового типа). (Возможно любое целое или вещественное значение, но строковое — не допускается!) Операции инкремента *увеличивают* значение операнда на единицу, декремента — *уменьшают* его на единицу. В качестве результата, *префиксные* операции возвращают значение операнда *после* его модификации, а *постфиксные — до* нее. Помимо этого, префиксные и постфиксные формы идентичны. Например:

```
a = 10;
```

```
  set:(a, 10) => 10;
```

```
b0 = ++ a; b1 = -- a;
```

```
  set:(b0, inc:a) => 11;
  set:(b1, dec:a) => 10;
```

```
c0 = a ++; c1 = a --;
```

```
  set:(c0, inc_p:a) => 10;
  set:(c1, dec_p:a) => 11;
```

В качестве исключения, значением *A* может быть (). Пустое значение считается эквивалентным нулю (т. е. инкремент превращает его в 1, декремент в -1).

Для быстрого сброса мутабельного целого *A* в 0, имеется операция **clr**(A) (не имеющая отдельной операторной формы). Как и операции инкремента/декремента, эта операция старается сохранить тип операнда (если он был вещественным числом, присваивается 0.0; а если пустым или целым — 0).

Операция обмена

Операция “обмена двух значений местами” полезна настолько часто, что для нее предусмотрен специальный функтор.

Обмен		
Синтаксис	***Нормальная форма***	***Семантика***
`V :=: W`	**`swap`**`(V, W)`	*Поменять местами значения V и W*

Разумеется, **swap** является мутатором, причем для обоих операндов: выражения *V* и *W* должны быть мутабельными (однако, они оба вычисляются только один раз).

```
[s1 s2] = ['alpha' 'beta'];
  set:((s1, s2), "alpha", "beta") => ("alpha", "beta");

s1 :=: s2;
  swap:(s1, s2) => ();

[s1 s2];
  (s1, s2) => ("beta", "alpha");
```

В качестве результата, **swap** всегда возвращает ().

Проверка на мутабельность

В языке есть встроенный функтор-предикат, проверяющий свой операнд на мутабельность:

Проверка мутабельности	
Нормальная форма	***Семантика***
`is_mut``(Q)`	*Является ли Q мутабельным?*

Если операнд *Q* мутабелен (например, переменная), результат — истина (1), в противном случае (например, если *Q* является скалярным значением), результат — ложь (0). Применение этой операции к простейшим выражениям языка (таким, как скаляры и строки) не имеет особого смысла — но дальше мы познакомимся с более сложными выражениями, для которых вопрос о мутабельности уже не так тривиален.

Поток управления

В большинстве традиционных (в основном, процедурных) языков программирования, для организации *потока управления* в программе используются специальные *инструкции* (или *операторы*). Их набор обычно жестко задан (и не расширяем обычными средствами языка).
В AWL реализован другой подход: для организации выполнения программы применяются встроенные функторы (мы назовем их *управляющими*). Это возможно потому, что в AWL, в общем случае, происходит *ленивая* передача параметров — то есть, передача параметра функтору не предполагает его вычисления/выполнения (в общем случае, последнее — *ответственность функтора*). Конечно, для многих функторов (например, для всех скалярных операций) вычисление параметров происходит *всегда* (т. к. скалярные функторы работают только с *результатами* своих операндов). Но для управляющих функторов это не верно: они могут вообще *не вычислять* какие-то свои операнды (тогда они являются *условными*), а могут вычислять их *многократно* (если они циклические, или *итераторы)*.
И более того: язык позволяет легко определять новые управляющие функторы (пользоваться которыми практически так же легко, как и встроенными). Инструменты для этого мы подробно рассмотрим несколько позже.

Условия и предикаты

Как уже говорилось выше — в языке не предусмотрено отдельного *логического* (или *булевского*) типа данных. Роль логических значений обычно играют целые числа (но в определенных случаях могут играть и значения других типов).
Когда в функторе *ожидается* условный операнд, в его роли может выступать практически любое значение. Оно интерпретируется таким образом: пустое значение (), скаляры 0, 0.0 или «» рассматриваются как *ложь* (**false**), все остальные значения — как *истина* (**true**). (Заметим, в частности, что строки «0» и «0.0» — в AWL также считаются *истинными*, так как они не пусты. В этом язык расходится с Perl.)
Когда функтор возвращает условный результат (т. е. является *предикатом*) — ограничения для него являются более строгими. В этом случае (исключая редкие специальные ситуации) возвращается целое **0** (если результат *ложен*), или целое **1** (если он *истинен*). Таким образом, результаты вычисления предикатов являются минимальным подмножеством всех «истинных» и всех «ложных» значений языка.
Функторов-предикатов в языке довольно много. К ним, в частности, относятся все операции числового сравнения (кроме **cmp**) и строкового сравнения (кроме **s_cmp**). Вот еще несколько полезных предикатов:

Принадлежность значения к диапазону	
Нормальная форма	***Семантика***
`inside``(Value, Range)`	*Истинно, если Value внутри диапазона Range*
`outside``(Value, Range)`	*Истинно, если Value вне диапазона Range*

Понятие «диапазона» мы уже рассмотрели ранее, рассказывая про получение отрезка строки. Обычно диапазон задается в виде *From..To*, что предполагает все целые числа от (*включая*) *From*, до (*не включая*) *To*. (Если задано одно число *V*, это синонимично 0..*V*.) Предикаты **inside** и **outside** компактнее, чем последовательность сравнений, и вычисляют значение *Value* только один раз. (К сожалению, они работают только с целыми числами: в языке не бывает «нецелых» диапазонов.)

Самыми тривиальными из всех предикатов являются нуль-арные «булевские литералы»:

Истина и ложь	
Нормальная форма	***Семантика***
`false``()`	*Всегда ложен (т. е. возвращает 0)*
`true``()`	*Всегда истинен (т. е. возвращает 1)*

На первый взгляд, их полезность не очевидна. Они чаще применяются там, где предикат требуется формально (например, в качестве «заглушки» для вычисления условия, истинность или ложность которого на самом деле известна заранее).

Условно-логические операции

Эти операции является своеобразной «промежуточной ступенью» между (чисто скалярными) логическими операциями, и условными операциями.

Условно-логические операции (бинарные):		
Синтаксис	***Нормальная форма***	***Семантика***
`~~ P`	**`c_not`**`(P)`	*Условно-логическое “не” (NOT):* *не P*
`P && Q`	**`c_and`**`(P,Q)`	*Условное логическое “и” (AND):* *если P — то Q, иначе 0*
`P || Q`	**`c_or`**`(P,Q)`	*Условное логическое “или” (OR):* *если P — то 1, иначе Q*

Они отличаются от своих скалярных аналогов (**and**, **or** и **not**) тем, что в качестве операндов допускается не только скаляры, но и (практически) любые типы данных. Операнд *P* всегда вычисляется как чисто логическое значение, но само вычисление операнда *Q* зависит от результата *P*. Если *P* является *истинным* (для **c_and**) или *ложным* (для **c_or**) — значение *Q* вычисляется (и возвращается как результат). В противном случае — **c_and** возвращает 0 (*ложь*), **c_or** возвращает 1 (*истину*). Легко видеть, что когда результаты *Q* также являются логическими (0 или 1) — **c_and** и **c_or** реализуют логические связки «И» и «ИЛИ» над результатами своих операндов. (Но при этом второй операнд может иметь произвольный тип, и вычисляется *факультативно*, в зависимости от результата первого).

Например, предикаты **inside** и **outside** из предыдущей главы — могут быть определены примерно таким образом:

- **`inside`**`(Value, From..To) = From <= Value && Value < To`
- **`outside`**`(Value, From..To) = Value < From || Value >= To`

Операция **c_not** также вычисляет истинность своего операнда, но возвращает противоположный результат: 0, если *P* истинно, или 1, если *P* ложно. (Другими словами, она реализует логику «НЕ».) В языке не очень часто возникает потребность в ней, т. к. для большинства условных операций имеются две симметричные формы: «*позитивная*» (с проверкой на истинность) и «*негативная*» (с проверкой на ложность).

Условные операции

В большинстве реальных программ приходится проверять какие-то условия — и, в

зависимости от результата, выполнять тот или иной вариант действий. В AWL *условное выполнение* обычно реализуется следующей группой тернарных функторов (как всегда, записываемых или в функциональной, или в синтаксической форме).

Условные операции (тернарные)		
Синтаксис	***Нормальная форма***	***Семантика***
`P ? Then : Else`	**`if`**`(P,Then,Else)`	*Проверка условия (позитивная): если P, то Then, иначе Else*
`P ~? Else : Then`	**`unless`**`(P,Else,Then)`	*Проверка условия (негативная): если не P, то Else, иначе Then*

Первый операнд (*P*) вычисляется *безусловно* (как логическое значение). Дальше вычисляется один из двух операндов (в зависимости от результата *P* и от *полярности* условия). Если *P* истинно, вычисляется/выполняется операнд *Then* (второй для **if** / третий для **unless**), в противном случае выполняется операнд *Else* (второй для **unless** / третий для **if**). В любом случае — вычисляется *только один* из операндов *Then* или *Else* (и его результат будет результатом всей операции).

Варианты **if** и **unless** практически взаимозаменяемы (по сути, они различаются только порядком второго и третьего операндов). Тем не менее, иметь две формы удобно — в т.ч. и потому, что (в функциональной форме) третий операнд может быть полностью *опущен* (т. е. операция может использоваться как бинарная). В этом случае, его выполнение является *пустой операцией*, возвращающей пустой результат (). (Собственно, могут быть опущены даже оба операнда — но в этом случае операция, кроме вычисления условия, вообще ничего не делает.)

Приведем примеры:

- ` Если v четно, то строка "чет", иначе "нечет" `
  ```
  v & 1 ? "нечет" : "чет";
  ```
- `Если x принадлежит отрезку, ограниченному a и b,
  то строка "<>"; если x меньше отрезка, то "<", иначе ">"`
  ```
  a < b ?
      (x < a ? "<" : x > b ? ">" : "<>")
      (x < b ? "<" : x > a ? ">" : "<>")
  ```

В своей синтаксической форме, условные операции имеют относительно низкий приоритет (на уровень ниже, чем логические связки). Таким образом, скобки вокруг условия, *Then* и *Else* обычно не нужны (но когда результат условной операции используется в выражениях, обычно требуются скобки вокруг самой операции).

Условно-логические операции (бинарные и унарные) — нетрудно выразить через условные операции (как в форме **if**, так и в форме **unless**):

- **c_and** (P, Q) = **if** (P, Q, 0) = **unless** (P, 0, Q)
- **c_or** (P, Q) = **if** (P, 1, Q) = **unless** (P, Q, 1)
- **c_not** (P) = **if** (P, 0, 1) = **unless** (P, 1, 0)

В этом смысле, их формально можно считать избыточными (что не отменяет их полезности).

Условные итераторы

Конечно, не менее важную роль, чем условия, в программах играют *циклы*. Очень часто требуются циклы с проверкой условия (выполняемой до или после выполнения тела цикла). В AWL такое реализуется следующими функторами:

Циклические операции с условием (бинарные):		
Синтаксис	***Нормальная форма***	***Семантика***
`P ?? Loop`	**`while`**`(P,Loop)`	*Цикл с предусловием (позитивным): пока P, выполнять Loop*
`P ~?? Loop`	**`until`**`(P,Loop)`	*Цикл с предусловием (негативным): пока не P, выполнять Loop*
`?? (Loop) P`	**`do_while`**`(P,Loop)`	*Цикл с постусловием (позитивным): выполнять Loop, пока P*
`~?? (Loop) P`	**`do_until`**`(P,Loop)`	*Цикл с постусловием (негативным): выполнять Loop, пока не P*

Как можно видеть, система циклических операций тоже имеет симметрию (причем двойную). Они различаются как *полярностью* условия (пока *истинно* / пока *ложно*), так и моментом его проверки (*перед* итерацией / *после* нее). Во всех формах выражение *P* является *условием цикла*, выражение *Loop* — *телом цикла*. Пока условие *P* истинно (для **while**-форм) или ложно (для **until**-форм) — тело цикла будет выполняться.
Нехарактерно (для других процедурных языков) и то, что все циклические операции также возвращают значение (которое может использоваться в дальнейших вычислениях). Этим значением будет результат вычисления *Loop* на последней итерации (или (), если *Loop* не было вычислено ни разу).
В заключение отметим, что при использовании функциональной формы — можно опустить операнд *Loop*. В этом случае, формы с предусловием/постусловием никак не различаются (**while**/**do_while** просто вычисляют *P* до тех пор, пока оно не станет ложным; а **until**/**do_until** — пока *P* не станет истинным). В обоих случаях — итератор вернет ().
Обратите внимание, что в синтаксической форме циклов с постусловием — тело цикла *Loop* должно иметь замкнутый синтаксис. Обычно это значит, что вокруг должны быть скобки (но не обязательно круглые, как предлагается в таблице).

Циклы с параметром

Достаточно часто возникает необходимость в другой разновидности циклов — с *параметром*, изменяющимся с фиксированным шагом на каждой итерации (как циклы **FOR** в Pascal, Modula или Oberon). В AWL присутствуют встроенные итераторы для реализации циклов с параметром (однако, они требуют, чтобы параметр был целым числом, и менялся только с шагом 1 или -1).

Циклические операции с параметром (тернарные):		
Синтаксис	***Нормальная форма***	***Семантика***
`?? V = (R)++ : Loop`	**`for_inc`**`(V, R, Loop)`	*Инкрементный цикл с параметром: для всех значений V в диапазоне R (по возрастанию) выполнить Loop.*
`?? V = (R)-- : Loop`	**`for_dec`**`(V, R, Loop)`	*Декрементный цикл с параметром: для всех значений V в диапазоне R (по убыванию) выполнить Loop.*

Итераторы с параметром имеют две формы: *инкрементную* и *декрементную*. В обоих

случаях, у них имеется три параметра:
- переменная (в общем случае *мутабельное выражение*) *V* (задающее *параметр цикла*);
- диапазон *R* (задающий *границы изменения* для параметра *V*);
- *тело цикла W*, выполняющееся на каждой итерации.

Формат *диапазона* мы уже рассматривали. Обычно диапазон задается в виде *From .. To*, где *From* является нижней границей (*включенной в него*), а *To* является нижней границей (*исключенной из него*). (Если в качестве диапазона задано единственное число *N*, то диапазон считается равным *0 .. N*.)

В зависимости от выбранной формы, параметр цикла меняет свое значение (для **for_inc**) от *From* до *To-1* с шагом 1, или же (для **for_dec**) от *To-1* до *From* с шагом -1. Для каждого значения параметра цикла — выполняется его тело *Loop*. Например, в случае:

```
for_inc (I, 10..15, Loop);
```

выражение *Loop* будет выполнено для всех значений `I = 10,11,12,13,14`. В случае:

```
for_dec (J, 50..100, Loop);
```

выражение *Loop* будет выполняться для всех значений `J = 99,98,97 ... 52,51,50`.

По завершении выполнения цикла с параметром — в качестве его результата возвращается последнее вычисленное значение *Loop* (как и для всех итераторов). Естественно, если диапазон является пустым (т.е. *From >= To*), то тело цикла не будет выполнено ни разу, а в качестве результата будет (). Значение параметра цикла после его выполнения следует считать неопределенным.

Заметим также, что параметру цикла можно присваивать любые значения в его теле, но это никак не повлияет на выполнение цикла (т. к. в начале каждой итерации цикла ему все равно присваивается предыдущее/следующее значение из диапазона). Сам диапазон также вычисляется один раз, перед началом выполнения цикла (его изменение во время выполнения цикла также ни на что не повлияет).

В заключение заметим, что иногда нужно выполнить определенный код заданное количество раз — но при этом за номером итерации следить не требуется. Это удобнее делать с помощью отдельного итератора **times**:

Цикл без параметра (бинарный):		
Синтаксис	***Нормальная форма***	***Семантика***
`?? N : Loop`	**`times`**`(N,Loop)`	*Выполнить Loop ровно N раз.*

Тело цикла (выражение *Loop*) выполняется ровно *N* раз (или ни разу, если `N <= 0`). Как и для всех итераторов, в качестве значения возвращается последнее вычисленное значение *Loop* (или (), если *Loop* не вычислялось).

Бесконечный цикл

Завершая разговор о функторах-итераторах, перечислим еще итератор, выполняющийся неограниченно (т. е. — теоретически — даже вечно).

«Бесконечный» цикл (унарный):		
Синтаксис	***Нормальная форма***	***Семантика***

«Бесконечный» цикл (унарный):		
	ever(Loop)	*Выполнять Loop снова и снова.*

Итератор **ever** вообще не имеет явного условия для своего завершения! Теоретически, он будет выполняться до конца работы программы, практически — пока не будет прерван другими способами (например, возбуждением исключения или вызовом **return**). В AWL вообще нет явных средств для выхода из тела итератора (аналогичных инструкции **break** в C или Java). Однако, аналогичного эффекта можно добиться с помощью *исключений* — но мы рассмотрим их позднее.

Простейший ввод/вывод

Практически любая реальная программа должна уметь вводить и выводить какие-то данные. До сих пор, весь ввод и вывод, с которым мы имели дело, осуществлялся *интерпретатором*, а не самой программой.

Однако, в AWL есть средства (а точнее, как всегда, встроенные функторы), обеспечивающие программе явный обмен информацией с внешней средой в процессе работы. Все функторы ввода/вывода оперирует понятием *потоков*, которые являются еще одним стандартным для AWL типом данных. Работа с потоками требует отдельного рассмотрения, но простейшие операции ввода и вывода стоит рассмотреть уже сейчас.

В рабочей среде AWL-среде всегда присутствуют три стандартных потока — *ввод*, *вывод* и *вывод диагностики*. Далее приведена сводка важнейших операций ввода-вывода.

Простейший вывод данных

Основным средством вывода данных является встроенный функтор **f_put**.

Операции вывода:		
Синтаксис	***Нормальная форма***	***Семантика***
`OUT <: V_LIST`	**`f_put`**`(OUT,V_LIST)`	*Вывести все значения из списка V_LIST в выходной поток OUT*
`<: V_LIST`	**`f_put`**`((),V_LIST)`	*Вывести все значения из списка V_LIST в поток стандартного вывода*

Выполнение выражения **`f_put`**`(Output, Expr)` (в операторном виде `Output <: Expr`) заключается в том, что выражение *Expr* (обычно это список, но возможен и скаляр) вычисляется, и его значение (или, последовательно, все значения из списка) выводятся в выходной поток, заданный аргументом *Output*. Если *Expr* является списком — элементы выводятся подряд и в заданном порядке (без каких-либо разделителей между ними). Если список содержит вложенные списки — они выводятся рекурсивно. В конце **f_put** также ничего лишнего не выводит (если после выведенных данных, например, требуется добавить конец строки, то символ «\n» нужно добавить в конец списка явно).

Если значением *Output* является (), то подразумевается в стандартный вывод. (В операторной форме — первый операнд () можно полностью опустить.)

Приведем простой пример вычислений, с наглядным выводом результата:

```
a = b = 2;
<: (a, '*', b, '=', a*b, '\n');
```

```
2*2= 4
```

Теперь мы также можем написать программу «Hello, world!»

```
<: "Hello, world!\n"
```

```
Hello, world!
```

Значением, возвращаемым **f_put**, является целое число: количество успешно выведенных скалярных элементов списка (не символов!)

Простейший ввод данных

Ввод данных является обратной операцией по отношению к их выводу. (Впрочем, «строгая» симметрия между операциями ввода и вывода невозможна, поэтому тут есть и серьезные различия).

Операции ввода:		
Синтаксис	***Нормальная форма***	***Семантика***
`IN :> M_LIST`	**`f_get`**`(IN,M_LIST)`	*Ввести значения из входного потока IN в список (мутабельный) M_LIST*
`:> M_LIST`	**`f_get`**`((),M_LIST)`	*Ввести значения из потока стандартного ввода в список (мутабельный) M_LIST*

Выполнение выражения **`f_get`**`(Input, Expr)` (в операторном виде `Input :> Expr`) заключается в том, что из входного потока *Input* читаются строки, и присваиваются элементам *Expr*. Если *Expr* — переменная, то читается и присваивается один элемент, если же это список из мутабельных элементов, то считывается и присваивается столько строк, сколько элементов имеется в списке. Заметим, что **f_get** всегда присваивает введенные значения в виде строк. (Если вам нужно преобразовать их, например, в числа — это придется делать самостоятельно.)

Если операнд *Input* равен (), или полностью опущен — входным потоком является стандартный ввод.

Строки, прочитанные с помощью **f_get**, *не содержат* в конце символа конца строки (в отличие, например, от аналогичных операций Perl). В качестве результата — **f_get** (аналогично **f_put**) возвращает количество успешно прочитанных аргументов. Если оно меньше, чем число запрошенных аргументов — обычно это означает, что был достигнут конец входного потока (или при вводе произошла ошибка).

Приведем несколько примеров совместной работы операций ввода и вывода. Вот тривиальный код, копирующий стандартный ввод на стандартный вывод (аналогично UNIX утилите **cat**). (Этот пример не 100% корректен: здесь если последняя строка файла не пуста, то при выводе она будет дополнена символом «\n»).

```
(:> inline) ?? (<: (inline, '\n'));
```

Вот несколько более интересный пример: мы читаем строки из стандартного ввода (накапливая их в списке *lines_list*), и затем выводим их в обратном порядке. (В этом примере используются списковые операции, описанные детально в следующей главе).

```
(:> inline) ?? (lines_list [<-] inline);
for_inc (n, #lines_list, <: (lines_list[n], '\n'));
```

Стандартные потоки

Теоретически, потоки ввода/вывода могут быть связаны с любыми файлами (или другими системными ресурсами). Но три потока, всегда доступных в AWL-системе, можно получить в явном виде, обращаясь к следующим нульарным функторам:

Стандартные потоки:	
Нормальная форма	***Семантика***

Стандартные потоки:	
f_in()	*Стандартный ввод интерпретатора.*
f_out()	*Стандартный вывод интерпретатора.*
f_err()	*Стандартный вывод ошибок интерпретатора.*

Заметьте, что эти функторы не требуют никаких аргументов, и единственной их задачей является обеспечение явного доступа к предопределенным системным потокам.

Вот так, например, можно вывести сообщение в стандартный поток ошибок:

```
f_err() <: ('Ошибка: значение i (', i, ') вне диапазона!');
```

Конкретное место, откуда берутся вводимые данные, и куда попадают выводимые, определяется операционной системой. Это может быть консоль, другие внешние устройства, дисковые файлы, сетевые соединения и т. д. Кроме того, во всех современных операционных системах присутствует возможность переназначения стандартного ввода и вывода перед запуском программы. В AWL присутствуют средства и для того, чтобы временно переадресовать стандартный ввод/вывод в процессе выполнения.
Этим, конечно, средства ввода и вывода далеко не исчерпываются. Но мы отложим их более подробное рассмотрение до главы о файлах.

Операции над списками

Как уже говорилось выше — *списки* AWL являются одной из самых фундаментальных структур данных. (Так, бинарные функторы принимают как аргумент список из двух элементов, а тернарные — из трех.) Однако, в языке определен и ряд операций, ориентированных на работу со списками как таковыми. Как и большинство прочих операций — они являются встроенными в язык функторами (имена которых обычно начинаются с **l_**). Для многих операций также доступен альтернативный синтаксис (символы для операций над списками обычно окружены квадратными скобками).
Любое значение, которое не является списком или (), мы далее будем называть *атомом*. (Например, все скаляры — атомарные значения, но атомы ими далеко не исчерпываются.) Рассмотрим списковые операции по порядку. (Далее *L* или *M* — операнды-списки.)

Длина списка

Длина списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`# L`	**`l_len`**`(L)`	унарная	*Длина списка L*

Унарная операция **l_len**(L) (или #L) всегда возвращает целое число — количество элементов в своем операнде-списке. Для любого нетривиального *закрытого* списка результат всегда больше 1. При применении к атому эта операция возвращает 1, а при применении к () — 0.

```
L_a = [10 20 30 40 50];
L_b = ['north' 'west' 'south' 'east'];
```

```
set:(L_a, 10, 20, 30, 40, 50) => (10, 20, 30, 40, 50);
set:(L_b, "north", "west", "south", "east") => ("north",
"west", "south", "east");
```

```
#L_a;
#L_b;
```

```
l_len:L_a => 5;
l_len:L_b => 4;
```

Как и прочие списковые операции языка — эта операция демонстрирует определенную общность свойств списков и атомов. Для нее атом равносилен списку длины 1, а () — списку длины 0. Можно сказать, что **l_len** возвращает общее количество “элементов” в своем операнде (невзирая на то, список это, атом или пустое значение).

Конкатенация списков

Конкатенация списков			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`L [+] M`	**`l_cat`**`(L, M)`	аддитивная	*Конкатенация списков L и M*

Бинарная операция **l_cat**(L, M) (в синтаксической форме `L [+] M`) *конкатенирует* (т. е.

последовательно сцепляет) два своих списка-операнда, и возвращает список-результат (содержащий сначала все элементы *L*, потом все элементы *M*). Например:

```
L1 = [11 22 33 44];
L2 = ['aaa' 'bbb' 'ccc'];
```

```
set:(L1, 11, 22, 33, 44) => (11, 22, 33, 44);
set:(L2, "aaa", "bbb", "ccc") => ("aaa", "bbb", "ccc");
```

```
L1 [+] L2;
L2 [+] L1;
```

```
l_cat:(L1, L2) => (11, 22, 33, 44, "aaa", "bbb", "ccc");
l_cat:(L2, L1) => ("aaa", "bbb", "ccc", 11, 22, 33, 44);
```

Любой из операндов может быть скалярным, или же пустым (). Сцепление списка с пустым значением (слева или справа) очевидно, ничего не меняет. В общем случае, для конкатенации списков верно такое очевидное тождество:

`#(L [+] M) == #(M [+] L) == #L + #M`.

Наверное, не надо пояснять различие между конкатенацией списков и применением конструктора списка. (Выражения L [+] M и (L, M) синонимичны только тогда, когда *L* является атомом.)

Заметим также, что первый операнд конкатенации копируется, а второй — *нет* (т. е. он содержится в результате в качестве «хвоста»). Однако, его можно скопировать явно с помощью **l_copy** (см. ниже).

Репликация списка

Репликация списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`L [*] N`	**`l_rep`**`(N, L)`	мультипликативная	*Репликация (повторение) N раз подряд списка L*

Бинарная операция **l_rep**(N, L) (в синтаксической форме: `L [*] N` — заметьте обратный порядок операндов!) *реплицирует* (последовательно повторяет) свой список-операнд *L* ровно *N* раз подряд, и возвращает список-результат. (Очевидно, что операнд *N* предварительно вычисляется как целое число.) Например (для списков из предыдущей главы):

```
L1 [*] 2;
L2 [*] 3;
```

```
l_rep:(2, L1) => (11, 22, 33, 44, 11, 22, 33, 44);
l_rep:(3, L2) => ("aaa", "bbb", "ccc", "aaa", "bbb", "ccc",
"aaa", "bbb", "ccc");
```

Операнд *L* также может быть атомом (в этом случае результат будет списком из *N* копий этого атома) или () (результат, очевидно, тоже будет (), независимо от значения *N*). Если N == 1, результатом будет просто *L*; если же N <= 0, то результатом всегда будет ().

Для списковой репликации также справедливо тождество:

```
#(L [*] N) == #L * N.
```

Также заметим, что (как и в случае конкатенации) результат также содержит операнд *L* в качестве своего «хвоста».

Инверсия списка

Инверсия списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`[~] L`	**`l_rev`**`(L)`	унарная	*Инверсия списка L*

Унарная операция **l_rev**(L) (в синтаксической форме `[~] L`) возвращает результатом *инверсию* своего операнда: список, содержащий все элементы операнда, но в противоположном порядке. Например (списки из предыдущих глав):

```
[~] L1;
[~] L2;
[~] 10;
```

```
l_rev:L1 => (44, 33, 22, 11);
l_rev:L2 => ("ccc", "bbb", "aaa");
l_rev:10 => 10;
```

Заметим, что инвертирование атомарного значения всегда возвращает его само (а инвертирование (), очевидно, также возвращает ()). Следующие очевидные тождества связывают *инверсию* списка, его длину, *конкатенацию* и *репликацию*:

```
#[~]L == #L
[~](L [+] M) [==] ([~]M) [+] ([~]L)
[~](L [*] N) [==] ([~]L) [*] N
```

Здесь операция-предикат [==] сравнивает операнды-списки на идентичность. Мы подробно рассмотрим ее позже.

Копирование списка

При присваивании списков происходит вычисление левого операнда. В большинстве случаев, если правый операнд является тривиальным (т. е. вычисляется сам в себя) — то он (для эффективности) присваивается левому без каких-либо изменений.
Но (как мы увидим ниже), списки (в отличие от скаляров) являются *мутабельными структурами* данных — то есть, значение списка после его создания может быть изменено. Таким образом, бывает существенно, работаем ли мы с «оригиналом» списка — или с его точной копией (в отличие от оригинала, копию можно безопасно изменять). Чтобы скопировать список, удобнее всего использовать следующую операцию:

Копия списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`[+] L`	**`l_copy`**`(L)`	унарная	*Точная копия списка L*

Эта операция просто возвращает копию списка *L*. Заметим, что если его элементами являются списки — они также копируются. С результатом можно безопасно делать все, что угодно, без риска испортить оригинал.
С помощью этой операции также легко реализовать «безопасные» варианты конкатенации (*L* [+] **l_copy**(*M*)) и репликации (**l_copy**(*L*) [*] *N*). Одна из причин, по которой эти операции не работают именно таким образом по умолчанию — это *эффективность*. Копирование списка (особенно длинного) требует времени (как и дополнительной памяти) — поэтому по умолчанию язык его избегает. В большинстве случаев это вполне оправдано, т. к. конкатенация, репликация или присваивание списка не предполагает его дальнейшего изменения. И только в тех (относительно редких) случаях, когда результат списковой операции может подвергнуться изменениям — для безопасности желательно применить к исходному списку **l_copy**.
Сама операцию копирования, фактически, работает, как конкатенация операнда и пустого значения:

```
[+](L) [==] (L [+] ())
```

Существует и обратная (в некотором роде) операция — получение *ссылки* на список:

Ссылка на список	
Нормальная форма	***Семантика***
l_ref(L)	*Ссылка на список L (вместо его копии).*

Эта операция предписывает как раз *не копировать* свой списковый операнд — но вернуть лишь ссылку на него. Однако, поскольку это именно та семантика, которой большинство списковых операций придерживается по умолчанию — в явном виде эта операция нужна довольно редко. Специальный синтаксис для нее не предусмотрен.

Поэлементный доступ к списку

Как уже подчеркнуто выше, списки являются мутабельными структурами данных. Простейший способ изменения списка состоит в изменении отдельных его элементов.

Доступ к элементам списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
L[N]	**l_item**(N,L)	унарная	*Элемент списка L с индексом N*

Обращение к **l_item** (N, L) (синтаксический эквивалент: L[N] имеет обратный порядок операндов) обеспечивает доступ к элементу списка *L* с *индексом N*. Индексы (как и в C) нумеруются с 0: первый элемент списка имеет индекс 0, последний — индекс #*L*-1. Индекс вычисляется как целое (с определенными для скаляров приведениями).
Это — *функтор доступа* (т. н. *аксессор*): он возвращает *мутабельный* (т. е. доступный для дальнейшего изменения) результат. Обращение к аксессору может использоваться в любом контексте, где требуется мутабельный операнд (такой, как переменная): как первый операнд присваивания, как операнд инкремента/декремента и пр. Важно заметить, что *аксессоры* и *мутаторы* — это разные категории функторов. (В отличие от вторых — первые, сами по

себе, ничего не меняют!) Конечно, любой аксессор может использоваться не только для изменения, но и просто для чтения тех данных, к которым он обеспечивает доступ.
Вот примеры работы с элементами списков из предыдущей главы:

```
L_a[2];
  l_item:(2, L_a) => 30;

L_b[3];
  l_item:(3, L_b) => "east";

++ L_a[1];
  inc:l_item:(1, L_a) => 21;

L_a;
  L_a => (10, 21, 30, 40, 50);

L_b[0] = 'north';
  set:(l_item:(0, L_b), "north") => "north";

L_b;
  L_b => ("north", "west", "south", "east");
```

Значение индекса может быть отрицательным. Тогда, оно рассматривается как номер элемента, отсчитываемый с конца (т. е. индекс -1 соответствует последнему элементу списка, индекс -2 предпоследнему элементу и т.д.)
Если значение индекса списка *L* некорректно (т. е. < -#L или >= #L), то операция возвращает (),.а попытка присвоить что-либо результату является *ошибочной*. В отличие от некоторых других языков (таких, как Perl), автоматически расширить список путем присваивания значения его несуществующим элементам *нельзя*. (Корректный способ сделать это мы покажем в следующей главе.)
Ради общности (о которой мы говорили выше) эта операция применима не только к спискам, но и к любым атомарным значениям. Именно, для любого атома *A* выражение A[0] возвращает его значение. Для любых других значений индекса возвращается ().

Изменение длины списка

Список является достаточно гибкой структурой данных. Например, его длину легко можно увеличить или уменьшить (но делать это необходимо явно).

Удлинить/укоротить список	
Нормальная форма	***Семантика***
`l_resize``(List, N)`	*Сделать длину списка List равной N.*

Прежде всего, эта операция является *мутатором*: она не создает новый список, но изменяет свой списковый операнд *List*! Значение *N* вычисляется как целое (как и для **l_item**). Если

длина списка *List* была меньше *N* — список *удлиняется* до длины *N*, дополняясь пустыми (содержащими значение ()) элементами. Если же она была больше *N* — список *укорачивается* до длины *N*. Результат операции всегда имеет длину *N*, и всегда является *открытым* списком (даже, если до операции *List* открытым списком не был).
Операция также применима к атомарному значению, и превращает его в открытый список, содержащий *N* элементов (все, кроме первого, будут пустыми). Наконец, применение его к переменной с пустым значением создает «пустой» список из *N* пустых элементов (его можно использовать как контейнер).
Напротив: если значение аргумента *N* <= 0 — то операция *полностью уничтожает* содержимое *List*, присваивая ему (). Впрочем, даже и частичное укорачивание списка является весьма деструктивной операцией: отрезанный «хвост» списка теряется безвозвратно. Поэтому использовать **l_resize** нужно с осторожностью.

Списки как бинарные деревья

Как и в LISP-подобных языках, каждый список фактически является рекурсивной структурой данных — *бинарным деревом*. Другими словами, каждый список состоит из двух компонент (условно называемых его "*головой*" и "*хвостом*"), причем каждой из этих компонент, в свою очередь, может быть список произвольной структуры. Для конструирования списка в явном виде, в языке имеется еще одна операция.

Примитивный списковый конструктор	
Синтаксис	***Семантика***
`Head::Tail`	*Создает список с «головой» Head и «хвостом» Tail.*

Так, при конструировании списка в простейшей форме — (A, B) — создается список, «головой» которого является A, а «хвостом» — B (то есть, эквивалент A :: B). Если в конструкторе списка более двух элементов, список строится рекурсивно: так, список (A, B, C, D) равносилен списку (A, (B, (C, D))) (или A :: B :: C :: D). Если список является «открытым», то для его последнего элемента «хвостом» является (). Например, открытый список (10, 11, 12,) равносилен 10 :: 11 :: 12 :: (). Такой подход к спискам альтернативен рассмотрению списка как линейного массива (и иногда открывает новые возможности для работы с ними).
Есть операции, явно позволяющие работать со списками как с бинарными деревьями.

Доступ к элементам списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`[<] L`	**`l_head`**`(L)`	унарная	*"Голова" списка L*
`L [>]`	**`l_tail`**`(L)`	унарная	*"Хвост" списка L*
	`l_head_by``(N, L)`		*Применить l_head N раз*
	`l_tail_by``(N, L)`		*Применить l_tail N раз*

Все эти операции также являются аксессорами. Операция **l_head**(L) предоставляет доступ к "голове" списка *L* (фактически, она эквивалентна L[0], и введена в основном для симметрии). Операция **l_tail**(L) предоставляет доступ к "хвосту" списка *L* (т. е. всем его элементам, кроме

первого). Эти операции аналогичны **car**/**cdr** в Lisp/Scheme, или **hd**/**tl** в ML. Обе операции также могут быть записаны в синтаксической форме: **l_head**(L) как [<] L, а **l_tail**(L) — как L [>]. (Если вы представите список в виде бинарного дерева, с корнем сверху, и потомками — снизу/слева и снизу/справа — то смысл этой мнемоники становится более понятен.)

Существуют также модификации этих операций, первым операндом которых является *повторитель* (вычисляемый как целое). Операция **l_head_by**(N, L) равносильна **l_head**^**l_head**^ ... **l_head**^(L), где **l_head** повторяется *N* раз. Совершенно аналогично, **l_tail_by**(N, L) равносильно **l_tail**^**l_tail**^ ... **l_tail**^(L), где **l_tail** повторяется *N* раз. Очевидно, что если $N == 1$, эти операции эквивалентны **l_head**(L) и **l_tail**(L); если же $N <= 0$, обе операции просто возвращают L. Эти операции позволяют эффективно работать с глубоко вложенными списками.

Как всегда, приведем несколько примеров (используя списки *L_a* и *L_b* из предыдущих глав):

```
[<] L_a;
L_a [>];
```

```
l_head:L_a => 10;
l_tail:L_a => (21, 30, 40, 50);
```

```
[<] L_b;
L_b [>];
```

```
l_head:L_b => "north";
l_tail:L_b => ("west", "south", "east");
```

```
l_tail_by(3, L_a);
l_tail_by(2, L_b);
```

```
l_tail_by:(3, L_a) => (40, 50);
l_tail_by:(2, L_b) => ("south", "east");
```

Поскольку все эти операции являются *аксессорами*, обращение к ним выдает мутабельный результат. В частности, присвоив вызову **l_tail** любой список, можно сделать его новым “хвостом” списка-операнда (старый хвост при этом может быть потерян!)

```
L_b [>] = ['west' 'south' 'east'];
```

```
set:(l_tail:L_b, "west", "south", "east") => ("west",
"south", "east");
```

```
L_b;
```

```
L_b => ("north", "west", "south", "east");
```

Как и следует ожидать, применение **l_head** к скаляру возвращает его самого, а применение к скаляру **l_tail** всегда возвращает () (т. к. у скаляров, в отличие от списков, нет “хвоста”).

У спискового конструктора «::» — более высокий приоритет, чем у вызова функтора (т. е. он связывает свои операнды сильнее). В частности, это означает, что запись типа **f**(a)::**g**(b) будет интерпретироваться не как список (**f**(a), **g**(b)) — а как **f**(a, **g**(b)). Для того, чтоб получить первую интерпретацию — вокруг второго операнда необходимы дополнительные скобки:

f(*a*)::(**g**(*b*)) равносильно (**f**(*a*), **g**(*b*)). Такой приоритет выбран для того, чтоб дать возможность записывать некоторые конструкции языка в более привычном (для процедурных языков) виде — прежде всего, это относится к условным и итеративным операциям.

Вместо ...	***… можно использовать:***
if (*Cond*, *Then*)	**if** (*Cond*):: (*Then*)
unless (*Cond*, *Else*)	**unless** (*Cond*):: (*Else*)
if (*Cond*, *Then, Else*)	**if** (*Cond*):: (*Then*):: (*Else*)
unless (*Cond*, *Else, Then*)	**unless** (*Cond*):: (*Else*):: (*Then*)
while (*Cond*, *Loop*)	**while** (*Cond*):: (*Loop*)
until (*Cond*, *Loop*)	**until** (*Cond*):: (*Loop*)
times (*Count*, *Loop*)	**times** (*Count*):: (*Loop*)

Таких вариантов возможно довольно много — в этой таблице приведены лишь самые употребительные.

Итераторы для списков

Для списков определены свои циклические операции (т. е. итераторы):

Списковые итераторы	
Синтаксис	***Семантика***
l_loop(V, List, Body)	*Для всех V из списка List (в прямом порядке) выполнить Body*
l_loop_r(V, List, Body)	*Для всех V из списка List (в обратном порядке) выполнить Body*

(Специальная синтаксическая запись для них не предусмотрена.)

Операции **l_loop** и **l_loop_r** тернарны, т. е. имеют три ожидаемых аргумента. Вызов **l_loop** (*V*, *List*, *Body*) выполняется следующим образом: для каждого элемента списка *List* его значение присваивается мутабельному *V* (переменной цикла), после чего вычисляется/выполняется выражение *Body* (тело цикла). Очевидно, что *Body* всегда будет выполнено *#List* раз. Результатом вычисления **l_loop** является результат *Body* на последней итерации цикла (или (), если список *List* был пуст). Например:

```
L1;
```

```
L1 => (11, 22, 33, 44);
```

```
l_loop (l, L1, <: ['<' l '>\n']);
```

```
<11>
<22>
<33>
<44>

l_loop:(l, L1, f_put:((), "<", l, ">
")) => 3;
```

Операция **l_loop_r** отличается от **l_loop** только тем, что проход по списку происходит *в обратном направлении*, от последнего элемента к первому. Например:

```
L2;
```

```
L2 => ("aaa", "bbb", "ccc");
```

```
l_loop_r (l, L2, <: ['<' l '>\n']);
```

```
<ccc>
<bbb>
<aaa>

l_loop_r:(l, L2, f_put:((), "<", l, ">
")) => 3;
```

Если вместо *List* задано атомарное выражение, то (как и для большинства других списковых операций) оно рассматривается как список из единственного элемента (т. е. *Body* выполняется один раз, с *V* = *List*). Если же значением *List* является (), то тело цикла *Body* не выполняется ни разу, и результатом также является ().
Заметим, что итераторы по списку работают эффективнее, чем (например) цикл **for_inc** или **for_dec** с «индивидуальным» доступом к каждому элементу списка по его индексу.

Вставка и удаление элементов списка

Довольно многие операции (например, фактически, все скалярные) рассматривают список как *стек* (на вершине которого находится начальный элемент списка). Абсолютно так же рассматривают списки и операции **l_push** и **l_pop**, которые позволяют добавлять элементы в его начало, или же удалять его начальные элементы. В принципе, эти операции избыточны (их можно заменить комбинациями списковых присваиваний), но их использование часто эффективнее и нагляднее.

Вставка/удаление элементов списка			
Синтаксис	***Нормальная форма***	***Приоритет***	***Семантика***
`L [<-] Q`	**`l_push`**`(L, Q)`	присваивания	*Добавить элементы из списка Q в начало списка L*
`L [->] Q`	**`l_pop`**`(L, Q)`	присваивания	*Извлечь элементы из начала L в список Q*

Конечно, обе эти операции являются мутаторами, т.к. изменяют свой первый операнд-список *L* (а операция **l_pop** изменяет и прочие свои операнды). Операция **l_push** (L, a, b, c, ...) (в синтаксической форме: *L* [<-] (*a*, *b*, *c*, ...)) *добавляет* в начало списка L результаты вычисления операндов *a*, *b*, *c* и т. д. Конечно, выражение L должно быть мутабельно. Заметим, что порядок, в котором они окажутся в L, *обратен* порядку, в котором они идут в списке операндов — т. е. начало *L* примет вид: (…, *c*, *b*, *a*, *L*). Если прежним значением *L* был атом, то он окажется в самом конце списка. Если же *L* имело пустое значение, то новым значением *L* просто станет список (..., *c*, *b*, *a*). Например (если *aa* ранее не было определено):

```
aa [<-] [1 2 3];
aa;
```

```
l_push:(aa, 1, 2, 3) => (3, 2, 1);
aa => (3, 2, 1);
```

```
aa [<-] ['xx' 'yy'];
aa;
```

```
l_push:(aa, "xx", "yy") => ("yy", "xx", 3, 2, 1);
aa => ("yy", "xx", 3, 2, 1);
```

Операция **l_pop** (L, ra, rb, rc, ...) (в синтаксической форме: L [->] (*ra*, *rb*, *rc*, ...)) выполняет обратную функцию: она извлекает из списка *L* элементы, последовательно присваивая их своим операндам *ra*, *rb*, *rc*, ... Естественно, эта операция требует, чтобы они, как и *L*, были мутабельны. Если после **l_pop** в *L* остается один элемент — *L* становится атомом. Если же из *L* извлекаются все элементы, значение L становится пустым. Вот и примеры (для предыдущего списка):

```
aa [->] I;
I; aa;
```

```
l_pop:(aa, I) => ("xx", 3, 2, 1);
I => "yy";
aa => ("xx", 3, 2, 1);
```

```
aa [->] [J K];
[J K]; aa;
```

```
l_pop:(aa, J, K) => (2, 1);
(J, K) => ("xx", 3);
aa => (2, 1);
```

```
aa [->] [L M];
[L M]; aa;
```

```
l_pop:(aa, L, M) => ();
(L, M) => (2, 1);
aa => ();
```

Можно использовать эти операции для вставки/удаления элементов не в начало, а в произвольную позицию списка (и даже в его конец), если использовать в качестве первого операнда обращение к **l_tail**/**l_tail_by**. Очевидно, что добавление элемента в начало «хвоста» списка равносильно его добавлению в соответствующую позицию списка. (Это же, конечно, справедливо и для удаления.) Например (для списка *L_b* из предыдущей главы):

```
L_b [<-] "northeast";
l_tail_by (2, L_b) [<-] "northwest";
l_tail_by (4, L_b) [<-] "southwest";
l_tail_by (6, L_b) [<-] "southeast";

L_b;
```

```
L_b => ("northeast", "north", "northwest", "west",
"southwest", "south", "southeast", "east");
```

Однако, для того, чтобы можно было добавлять элементы в самый конец списка — список должен быть *открытым* (т. к. у закрытого списка просто нет изменяемой конечной позиции после последнего элемента). Впрочем, заметим, что добавление элементов в конец списка — более медленная операция, чем добавление их в начало (и тем более медленная, чем длиннее этот список). Вот почему при создании списка элементов неизвестной заранее длины (например, при чтении строк из файла или стандартного ввода) — обычно быстрее сначала прочитать все элементы в пустой список (с помощью **l_push**), а потом инвертировать результат (с помощью **l_rev**), чтобы элементы в нем шли в оригинальном порядке.

Рассечение списка

С помощью конкатенации (**l_cat**) легко создать один список из двух. Существует и обратная операция (**l_split**) — *рассечение* списка на его начальный и конечный фрагмент. Но (в отличие от конкатенации) эта операция не определена однозначно — поскольку список можно разрезать в разных местах, она требует дополнительный аргумент, указывающий, где именно список-операнд разрезается.

Рассечение списка в заданной позиции	
Нормальная форма	***Семантика***
l_split(N, List)	*Разрезать список List в позиции N.*

Операция **l_split** является мутатором для своего спискового операнда *List*. Она преобразует его в список из двух элементов: начального фрагмента длиной в *N*+1 элемент, и конечного (с оставшимися элементами). Приведем пример:

```
List = ("northeast", "north", "northwest", "west",
"southwest", "south", "southeast", "east");
l_split (3, List);
```

```
l_split:(3, List) => (("northeast", "north", "northwest",
"west"), "southwest", "south", "southeast", "east");
```

Как видим, применение этой операции разрезает список на начальный фрагмент (голову) и конечный (хвост), причем элементов в начальном фрагменте списка *всегда на 1 больше*, чем задано аргументом *N*. В частности, это означает, что если $N == 0$, то список *List* не будет изменен вообще (почему?). Если значение *N* отрицательно, или больше длины списка, то операнд также не меняется.

В заключение заметим, что в языке намного больше встроенных функторов, работающих со списками. Однако, большинство из них требует в качестве аргумент(ов) *ссылки* на другие функторы (к рассмотрению которых мы и переходим).

Пользовательские функторы

Как мы видели, в языке AWL изрядная часть вычислительной работы выполняется функторами. Однако, все функторы, с которыми мы имели дело выше — были *предопределенными* (т.е. встроенными в систему). Но пользователь легко может определить и свои *собственные* функторы (работа с которыми в большинстве случаев не сложнее, чем с предопределенными). Это — основной механизм, реализующий расширение языка.

Декларация функтора

Описание (декларация) пользовательского функтора (в самой общей форме) выглядит так:

```
! functor (par1 par2 par3 … parM) : [loc1 loc2 loc3 … locN] =
functor_body
```

Здесь:

functor — имя (идентификатор) описываемого функтора;
par1 … parM — идентификаторы *параметров* нового функтора;
loc1 … locN — идентификаторы *локальных переменных* нового функтора;
functor_body — тело функтора.

Обязательными элементами в этом описании являются только *имя функтора* и его *тело*. Если у описываемого функтора нет параметров (т. е. он является *нульарным*) — то список параметров может быть полностью опущен (вместе с окружающими его круглыми скобками). Если у функтора нет локальных переменных — их список также может быть опущен (вместе с окружающими квадратными скобками и предшествующим двоеточием). Тело функтора должно быть непустым, им может быть *произвольное* AWL-выражение (на практике, им чаще всего является блок). Им даже может быть пустое значение () или [], или же пустой блок {} — это вполне законно (правда, пользы от такого функтора немного).
Заметим, что разделителями в списках параметров и локальных переменных являются пробельные символы, или их последовательности. Запятые здесь не требуются!
Декларация функтора создает свою собственную область видимости (т. н. *пространство имен*), в которой локализуются все его параметры (*par1 … parN*) и локальные переменные (*loc1 … locN*). По умолчанию они невидимы за пределами функтора, но доступны в его теле. В своей области видимости локальные переменные временно замещают одноименные внешние (если последние имеются). Любая переменная, не декларированная в каком-то из этих двух списков — рассматривается как *глобальная*.
Приведем примеры описаний простых функторов:

```
` Диагональ параллелепипеда (со сторонами x,y,z) `
! Diag(x y z) = sqr(x*x + y*y + z*z);

` Факториал от N (итеративное вычисление) `
! Fact(N) : [F] = { F = 1; N ?? (F =*: N --); F };

` Сцепление двух строк, в скобках и через ':' `
! Join2 (s1 s2) = '(' +$ s1 +$ ':' +$ s2 +$ ')';
```

Как результат обработки перечисленных описаний, интерпретатор выдаст следующее:

```
! Diag:(x y z) [3] = sqr:add:(add:(mul:(x, x), mul:(y, y)),
```

```
mul:(z, z)) => Diag:;

! Fact:(N) [2] = {
set:(F, 1);
while:(N, comb:mul:(F, dec_p:N));
F
} => Fact:;

! Join2:(s1 s2) [2] = s_cat:(s_cat:(s_cat:(s_cat:("(", s1),
":"), s2), ")") => Join2:;
```

Обратите внимание, что при выводе заголовок функтора дополнен числом в квадратных скобках, указывающим размер его *локального контекста* (сумма количества его параметров и локальных переменных). Кроме того, каждое описание функтора также возвращает (не используемое) значение — *ссылку* на новый функтор (о ней в подробностях далее).

Вызов функтора

Использовать определенные пользователем функторы обычно почти так же просто, как и встроенные в язык. Например:

```
Diag(10, 20, 30);
  Diag:(10, 20, 30) => 37.416574;

Fact(6);
  Fact:6 => 720;

Join2 ('Hello', 'World');
  Join2:("Hello", "World") => "(Hello:World)";
```

Вот как это выглядит с технической точки зрения: при вызове пользовательского функтора выполняются следующие действия:

- создается *новый локальный контекст*, с которым связываются все параметры и локальные переменные функтора;
- параметры функтора инициализируются *списком аргументов*, заданных в его вызове. (У инициализации есть свои нюансы — подробнее об этом в следующей главе.) Все не инициализированные явно параметры и локальные переменные — получат значение ();
- вычисляется/выполняется *тело функтора*;
- *уничтожается локальный контекст*, вместе со значениями всех переменных/параметров;
- наконец, вычисленное значение тела функтора *возвращается в качестве результата*.

Вот еще несколько практически полезных функторов:

```
` Среднее арифметическое чисел в списке L `
! avg_arithm (L):[S V] = {
    S = 0;
    l_loop (V, L, S =+: V);
    S / #L };
```

```
! avg_arithm:(L) [3] = {
```

```
  set:(S, 0);
  l_loop:(V, L, comb:add:(S, V));
  div:(S, l_len:L)
  } => avg_arithm:;
```

```
` Среднее геометрическое чисел в списке L `
! avg_geom (L):[P V] = {
     P = 1;
     l_loop (V, L, P =*: V);
     exp_by (P, 1/#L) };
```

```
  ! avg_geom:(L) [3] = {
  set:(P, 1);
  l_loop:(V, L, comb:mul:(P, V));
  exp_by:(P, div:(1, l_len:L))
  } => avg_geom:;
```

```
` Строковое сцепление всех элементов в списке L
  (со вставкой между ними строк-разделителей S) `
! l_join (L S):[R V] = {
     R = string(l_head(L));
     l_loop (V, l_tail(L), R =+$: (S +$ V));
     R
     };
```

```
  ! l_join:(L S) [4] = {
  set:(R, string:l_head:L);
  l_loop:(V, l_tail:L, comb:s_cat:(R, s_cat:(S, V)));
  R
  } => l_join:;
```

Принципиальная разница между локальными и глобальными переменными заключается в том, что глобальные переменные существуют в *единственном* экземпляре (а время их жизни, фактически, совпадает со временем выполнения программы). Локальные переменные (равно, как и параметры функтора) могут вообще *не существовать* (когда данный функтор не вызван), а могут и существовать во множестве экземпляров (так как функтор может быть вызван и рекурсивно). Можно сказать, что локальные переменные и параметры *привязаны* к своему функтору, и не существуют в отрыве от него.

Передача параметров

Единственное различие между параметрами функтора и его локальными переменными состоит в том, что параметры *инициализируются* при вызове функтора (в то время, как переменные просто очищаются в ()). (Далее, список значений, передаваемых функтору при вызове, мы будем называть *списком аргументов*, чтобы отличать его от параметров.) Фактически, семантика передачи аргументов функтору мало отличается от семантики спискового присваивания: происходит примерно то же самое, что и при выполнении выражения:

```
(par1, par2, …, parN) = (arg1, arg2, …, argM);
```

(где *par1 ... parN — параметры* функтора, а *arg1 ... argM — аргументы* при его вызове). И, хотя семантику присваивания списков мы уже рассматривали — здесь на ней стоит

остановиться еще раз.
Если длина списка параметров *совпадает* с длиной списка аргументов — никаких особых сюрпризов не предвидится. Они просто присваиваются поэлементно — т.е. *arg1* присваивается *par1*, *arg2* присваивается *par2* и т. д. до самого конца. Заметим, что каждый аргумент функтора сам тоже может быть списком — в этом случае, они и присваиваются как списки, с сохранением своей внутренней структуры.
Если список аргументов *длиннее* списка параметров — все параметры получат значения аргументов (по порядку), кроме последнего, значением которого станет *весь* оставшийся не присвоенным «хвост» списка. Например: если функтор с параметрами (a, b, c) получит список аргументов (10, “abc”, 20, “def”) — результатом будет a = 10, b = ”abc” и c = (20, “def”). Применяется тот же принцип, что и при присваивании списков — значения элементов не должны «пропадать».
Поэтому, в AWL очень легко определять функторы с переменным числом аргументов. Например, мы можем определить функтор **printf** примерно таким образом:

```
! printf (format arguments) = { … };
```

(Полного описания не будет, но идея ясна.) Если он вызван таким образом:

```
printf (“Name = %s, Age= %u”, user_name, user_age);
```

то параметр *format* получит строку “Name = %s, Age= %u”, а параметр *arguments* — список из двух элементов (user_name, user_age). Далее уже сам функтор **printf** легко может извлекать элементы этого списка (и форматировать в соответствии со спецификаторами, включенными в строку *format*).
Еще одна применение для списков аргументов переменной длины — передача их *другим функторам*. Например, нетрудно создать функтор-*диспетчер*, вызывающий один из нескольких функторов, в зависимости от значений первого (или даже нескольких первых) параметров — и передающий им все остальные (при этом про содержимое списка «остальных» диспетчер может вообще ничего не знать, и передавать их одним-единственным списковым аргументом).
Наконец, если список аргументов *короче* списка параметров — то всем параметрам, которым не хватит значений в списке аргументов, будет просто присвоено пустое значение (). (Мы предполагаем, конечно, что функтор позволяет опускать значения параметров, когда они не нужны — иначе, это уже будет ошибкой.)
Как уже говорилось, аргументами функтора могут быть данные любого типа — скаляры, списки, нетривиальные структуры данных, которые мы рассмотрим позже. Неизменяемые данные (например, скаляры) передаются исключительно *по значению*, а *мутабельные* (т. е. большинство прочих) — *по ссылке*. Поэтому, с изменением последних надо быть осторожнее (считайте, что функтор работает с оригиналом данных, если он явно не сделал их копию).
Наконец, существует еще один очень важный тип передачи параметров — т. н. «ленивая» передача. Но разговор об этом мы отложим до главы о «ленивых вычислениях».
В завершение заметим, что функтор может не только принимать, но и *возвращать* значения любого типа — например, списки:

```
! rotate (angle x y) =
(x*cos(angle) - y*sin(angle), x*sin(angle) + y*cos(angle));
```

Этот функтор поворачивает свой вектор-операнд *(x, y)* на угол *angle* (в радианах), и возвращает вектор-результат. Его можно использовать, например, так:

```
(X, Y) = rotate (pi(1/3), X, Y)
```

Рекурсия

Пользовательские функторы могут быть *рекурсивными*, т. е. они могут обращаться к самим себе — как прямо, так и косвенно. Для прямого обращения не требуется никаких специальных усилий — имя определяемого функтора *сразу* доступно в его теле. Например, вот рекурсивное определение факториала (итеративное мы уже привели выше):

```
! r_fact (N) = N ? N * r_fact (N-1) : 1;
```

```
! r_fact:(N) [1] = if:(N, mul:(N, .r_fact:sub:(N, 1)), 1)
=> r_fact:;
```

```
r_fact (10);
```

```
r_fact:10 => 3628800;
```

Как мы уже говорили, у *каждого* вызова рекурсивного функтора есть свой собственный *контекст* (набор аргументов и локальных переменных), независимый от других обращений к нему. Один рекурсивный вызов также может порождать несколько других. Например, вот так (алгоритм треугольника Паскаля) можно рекурсивно вычислять биномиальные коэффициенты (т. е. число всех комбинаций из N по M):

```
! r_comb (N M) = (0 < M && M < N) ?
r_comb (N-1, M) + r_comb (N-1, M-1) : 1;
```

```
! r_comb:(N M) [2] = if:(c_and:(lt:(0, M), lt:(M, N)), add:
(.r_comb:(sub:(N, 1), M), .r_comb:(sub:(N, 1), sub:(M,
1))), 1) => r_comb:;
```

```
r_comb (8, 3);
```

```
r_comb:(8, 3) => 56;
```

Факториалы и биномиальные коэффициенты легко могут быть вычислены и не рекурсивным путем, но бывают ситуации, когда обойтись без рекурсии весьма затруднительно. Известный пример — *функция Аккермана*, определенная для двух неотрицательных аргументов. При ее вычислении — даже для небольших значений аргументов происходит большое количество вложенных вызовов.

```
! ack (m n) = m ? (ack (m-1, n ? ack (m, n-1) : 1)) : n+1;
```

```
! ack:(m n) [2] = if:(m, .ack:(sub:(m, 1), if:(n, .ack:(m,
sub:(n, 1)), 1)), add:(n, 1)) => ack:;
```

```
ack (3, 5);
```

```
ack:(3, 5) => 253;
```

(В этом определении мы для наглядности опустили проверку валидности аргументов *m* и *n* — в реальной программе она *должна* присутствовать.)

Рекурсию, как и любой мощный механизм вычислений, следует применять с большой

осторожностью! Только на программисте лежит ответственность за то, чтобы рекурсивный вызов корректно завершился. (Если же рекурсивный алгоритм все-таки зациклится — результатом этого обычно является почти немедленный крах программы, вызванный переполнением стека вычислений.)
Рекурсию также следует избегать в случаях, где вместо нее легко можно использовать итеративный алгоритм — они обычно работают эффективнее. (Некоторые языки содержат встроенные средства оптимизации хвостовой рекурсии в итерацию — но не AWL.)

Семейства функторов

Также нередка ситуация, когда нужно создать несколько *взаимно-рекурсивных функторов* (т. е. таких, каждый из которых прямо или косвенно обращается к остальным.) Здесь, однако, имеется тонкость: язык требует, чтобы декларация каждого функтора *предшествовала* его первому использованию. Но в группе функторов, связанных взаимной рекурсией — хотя бы один из них должен быть вызван *до того*, как будет корректно декларирован. Для решения этой проблемы имеется специальный языковый механизм — *семейства функторов*.
Общий синтаксис описания семейства функторов является *обобщением* для описания одиночного функтора:

```
! { functor1 functor2 functor3 … functorM } = {
     (par11 … par1N) : [loc11 … loc1N] = body1,
     (par21 … par2N) : [loc21 … loc2N] = body2,
     …
     (parM1 … parMN) : [locM1 … locMN] = bodyM
     }
```

Описание состоит из двух частей: *заголовочной* (перечисляющей имена описываемых функторов) и собственно *реализационной*. Между ними всегда стоит знак «=».Здесь:

functor1 … functorM — идентификаторы всех описываемых функторов;
parn1 … parnN и locn1 … locnN — списки *параметров и локальных переменных* (для каждого из *M* описываемых функторов они свои);
body1 … bodyM — тела (для каждого из M описываемых функторов).

Обратите внимание на то, что списки имен функторов, и их реализаций заключены в *фигурные* скобки. Имена функторов в списке разделяются *пробельными символами*, а их реализации — *запятыми*. Длины обоих списков должны совпадать!
Основным преимуществом описания нескольких функторов в виде «семейства» является то, что имя каждого функтора становится доступным *сразу после* заголовка (т.е. еще до того, как его описание буде завершено). Таким образом, *каждый* из описываемых функторов семейства может беспрепятственно обращаться к любому из других, перечисленных в заголовке (включая, разумеется, и себя). Помимо несколько усложненного синтаксиса описания, никаких других отличий «семейственного» функтора от обычного не существует. Поэтому, в семейство можно объединить описание произвольной группы функторов, в том числе и не связанных взаимной рекурсией — но именно в последнем случае их объединение наиболее оправдано.
Приведем пример семейства из двух функторов (**is_even** и **is_odd**), вычисляющих четность или нечетность произвольного целого числа (для чего каждый из них рекурсивно обращается к другому).

```
! { is_even is_odd } = {
    ` is_even => `
    (n) = abs(n) <= 1 ? n == 0 : is_odd  (n > 0 ? n-1 : n+1),

    ` is_odd => `
    (n) = abs(n) <= 1 ? n <> 0 : is_even (n > 0 ? n-1 : n+1)
    };
```

```
{
! is_even:(n) [1] = if:(le:(abs:n, 1), eq:(n, 0),
.is_odd:if:(gt:(n, 0), sub:(n, 1), add:(n, 1)));
! is_odd:(n) [1] = if:(le:(abs:n, 1), ne:(n, 0),
.is_even:if:(gt:(n, 0), sub:(n, 1), add:(n, 1)));
} => (is_even:, is_odd:);
```

Это, конечно, не вполне обычный способ проверки на четность — но он тоже работает:

```
for_inc (i, -3..4, <: (i, ': ', is_even(i), ', ', is_odd(i),
'\n'));
```

```
-3: 0, 1
-2: 1, 0
-1: 0, 1
0: 1, 0
1: 0, 1
2: 1, 0
3: 0, 1
for_inc:(i, (neg:3, 4), f_put:((), i, ": ", is_even:i, ",
", is_odd:i, "
")) => 6;
```

Конечно, такой способ проверки четности, мягко выражаясь, неэффективен. В реальной программе лучше использовать что-нибудь вроде:

```
! is_even(n) = n %% 2 == 0;
! is_odd(n) = n %% 2 <> 0;
```

Локальные функторы

Описания функторов могут быть *вложенными*: т. е. в теле любого функтора может быть декларировано любое число других функторов. Как и для параметров и локальных переменных — эти описания *локализуются* в данном функторе, и недоступны снаружи (если только не использовать *квалификаторы*, подробное рассмотрение которых есть в главе о классах). При этом (как и для глобальных переменных и функторов) пространства имен для локальных переменных и для функторов являются *независимыми*. Так что, говоря выше о собственном пространстве имен функторов, мы все слегка упрощали — т. к. на самом деле, таких пространств целых *два*. Переменные (параметры) и локальные функторы принадлежат к двум *разным* пространствам имен (которые никогда не пересекаются друг с другом, и не вступают в конфликты).

Каждому локальному функтору неявно доступны все локальные переменные и параметры того функтора, в который он вложен. Кроме того, он может вызывать локальные функторы, декларированные до него. В этом отношении AWL похож на другие языки, в которых

поддерживается локальность функций (например, на Паскаль). (Однако, заметим, что если внешний функтор является рекурсивным, и был вызван более одного раза, то внутренним функторам будут доступны только переменные *последнего* (вложенного глубже всех) из его вызовов.)

На вложенность описаний функторов также не накладывается никаких ограничений: локальные функторы могут иметь свои собственные локальные функторы, и так далее, на практически неограниченную глубину.

Использование локальных функторов поощряется. Правильная локализация функторов уменьшает загрузку таблиц имен, улучшает структуру программного кода, и может даже уменьшить количество передаваемых параметров. Т. к. каждому вложенному функтору неявно доступен весь контекст внешнего — явно передавать ему многие параметры часто нет нужды.

Приведем (довольно простой) пример локального описания функтора. Функтор **permute** работает как *генератор перестановок*: вызов **permute**(N) находит все N! перестановок целых чисел от 0 до *N*-1 (и выводит каждую из них на консоль). Заметим, что сам рекурсивный перебор перестановок выполняет *локальный функтор* **r_perm** (которому доступны и все локальные переменные **permute**, такие, как *N* и *perm*). Для перестановки элементов в списке *perm* активно используются итераторы с инкрементом (**for_inc**) и декрементом (**for_dec**), как и операция обмена «:=:».

```
` Перестановки чисел, диапазон 0..N `
! permute (N):[perm] = {
` Текущая перестановка `
perm = 0 [*] N;

` Рекурсивный перебор `
! r_perm (n):[i] =
    n <> N ? {
          perm[n] = n;
          n ++;
          for_dec (i, 1..n, perm[i] :=: perm[i-1]);
          r_perm (n);
          for_inc (i, 1..n, {
               perm[i] :=: perm[i-1];
               r_perm (n)
               })
          }
     :
     ` Конец рекурсии: вывести эту перестановку `
    {
     <: '[ ';
     for_inc (i, N, <: (perm[i], ' '));
     <: ']\n';
     };

` Входная точка рекурсии `
r_perm (0);
};
```

Работает **permute** следующим образом:

```
permute (3);
```

```
[ 2 1 0 ]
[ 1 2 0 ]
[ 1 0 2 ]
[ 2 0 1 ]
[ 0 2 1 ]
[ 0 1 2 ]
permute:3 => ();
```

Заметьте, что из-за особенностей этого алгоритма самой первой всегда выводится *инверсная* перестановка (числа от N-1 до 0 в обратном порядке), а *прямая* перестановка выводится самой последней. Но, хотя **permute** находит все перестановки, но в таком виде от него не очень много пользы — он умеет только выводить их на консоль. Его можно сделать куда более полезным — например, если вместо этого он будет применять к перестановке произвольный функтор, заданный параметром-*ссылкой*.

Ссылки на функторы и косвенные вызовы

Все рассмотренные ранее вызовы функторов были *прямыми* — т. к. имя вызываемого функтора в них всегда задавалось явно. Однако, может возникнуть необходимость обратиться к функтору, который определяется только в процессе выполнения программы. Такой вызов функтора называется *косвенным*, и он возможен через *ссылку* на него.
Ссылки на функторы — это новый (ранее не рассмотренный) тип языка. Для того, чтобы получить ссылку на уже имеющийся функтор, предусмотрена специальная нотация:

```
! my_func
```

Значение этого выражения — *ссылка на функтор* с именем **my_func** (который может быть как встроенным, так и определенным пользователем). Заметим, что без предшествующего восклицательного знака, это станет обращением к *переменной* под названием *my_func* (даже если она не существует — она будет создана, как глобальная и пустая). Различие между идентификаторами функторов и переменных заложено в синтаксис языка — поэтому, если мы имеем дело с функтором, это должно быть указано явно!
Это значение может быть присвоено переменной (или любому другому мутабельному — например, элементу списка). Как и любой атомарный тип данных, его можно также передавать функторам как параметр, или возвращать из них как значение. Но, помимо всего этого, ссылку на функтор можно еще *вызвать*, передав ей аргумент (или их список).

Косвенный вызов функтора		
Синтаксис	***Нормальная форма***	***Семантика***
`Func ! Args`	**apply**`(Func, Args)`	*Вызвать ссылку на функтор Func с аргументами Args*

Вызов функтора (в любой из этих форм) сам по себе является обращением к *встроенному* функтору **apply**, первым аргументом которого является ссылка на вызываемый функтор, а следующим(и) — его аргумент(ы). (Если **apply** вызвано с единственным аргументом, то вызываемому функтору передается *пустой* список аргументов.) В качестве результата — **apply** возвращает результат вычисления вызванного функтора. Самый первый аргумент для **apply** является *обязательным*, и должен быть ссылкой на функтор — в противном случае,

сообщается об ошибке (и никакого вызова, конечно, не происходит).
Помимо того, что вызов через **apply** является *косвенным* — он *ничем* не отличается от аналогичного прямого вызова: аргументы обрабатываются так же, и возвращается тот же результат. Другими словами, традиционный вызов **any_func** (arg1, arg2, … argN) *всегда* равносилен **apply** (! **any_func**, arg1, arg2, … argN) — он имеет тот же эффект, и возвращает тот же результат. Например, чтобы перемножить числа X и Y, можно написать X*Y или **mul** (X, Y) — а можно и (!**mul**) ! (X, Y). Правда, косвенный вызов, конечно, менее эффективен, чем прямой — поэтому нет смысла вызывать таким образом однозначно определенный функтор. Но вот в ситуации, когда вызываемый функтор *может меняться* — альтернативу косвенным вызовам трудно отыскать. Например, выражение:

```
(!add, !sub, !mul, !div) [index] ! (A, B)
```

возвращает сумму, разность, произведение или частное двух аргументов *A* и *B*, в зависимости от значения переменной *index* (предполагается, что последнее находится в диапазоне 0..4).
Косвенный вызов допустим для любых функторов, не только для скалярных. Например, как мы помним, все структуры управления (условия, итераторы) — это тоже встроенные функторы, которые тоже можно вызывать через **apply**.
Предположим, например, что в программе требуется выполнять одну из последовательностей действий *ActionX* и *ActionY*, в зависимости от некоего условия *Test*. Кроме того, сам критерий «успешности» проверки тоже меняется: если значение переменной *phase* истинно, то при истинности *Test* выполняем *ActionX*, иначе *ActionY*. Если же *phase* ложно — то поступаем наоборот. Это можно реализовать по-разному — например, так:

```
(phase ? !if : !unless) ! (Test, ActionX, ActionY)
```

Другой пример: предположим, что надо выполнить итератор для некоторого диапазона целых чисел *Range* — однако, направление их перебора (убывание или возрастание) должно определяться переменной *direction* при выполнении.

```
(direction ? !for_dec : !for_inc) ! (I, Range, Loop)
```

Собственно, это эквивалент следующего кода:

```
(direction ? for_dec(I,Range,Loop) : for_inc(I,Range,Loop))
```

Однако, код выше намного компактней, особенно если в *Loop* выполняются нетривиальные действия. К тому же, если весь код *Loop* находится в одном месте, уменьшается вероятность случайных ошибок, когда в одном месте в *Loop* вносятся изменения, а в другом про это забывают.
Наконец, мы приведем пример функтора, получающего ссылку на функтор как аргумент. Для функтора **reduce** семантика аналогична встроенной операции [=]: т. е. он «сворачивает» список *L* (свой второй операнд) путем последовательного применения бинарного функтора-аргумента *op* к каждой паре элементов (до тех пор, пока список *L* не будет исчерпан).

```
! reduce (op L) : [val i] = {
    val = L[0];
    for_inc (i, 1..#L, val = op ! (val, L[i]));
    val };
```

Функтор **reduce** работает примерно так же, как и встроенная в язык редукция (хотя и

медленнее, конечно).

```
reduce (!add, [1 2 3 4 5]);
```

```
reduce:(add:, 1, 2, 3, 4, 5) => 15;
```

```
reduce (!mul, [1 2 3 4 5]);
```

```
reduce:(mul:, 1, 2, 3, 4, 5) => 120;
```

(Однако, в отличие от операции [=], **reduce** вполне может быть применен и к не скалярным, и даже к пользовательским функторам.)

Анонимные функторы

Механизм ссылок на функторы предоставляет еще одну интересную возможность: создания *анонимных* (безымянных) функторов прямо там, где нужна ссылка на них. Таким образом, если функтор требуется *только один раз* — нет нужды декларировать его отдельно и придумывать для него уникальное имя. В функциональных языках подобные конструкции называются *лямбда-выражениями*.

Синтаксически, анонимный функтор является не самостоятельным описанием, а *выражением*, с приоритетом унарных операций и следующим синтаксисом:

```
! (par1 par2 … parN) : [loc1 loc2 … locN] = functor_body
```

Другими словами, описание анонимного функтора практически аналогично описанию обычного. Но есть два основных различия: у анонимного функтора *отсутствует* имя, а его тело *должно* быть синтаксически замкнутым выражением. (На практике это обычно означает, что вокруг него должны быть скобки любого типа — круглые, квадратные или фигурные.) Как и для обычных функторов, списки параметров и локальных переменных могут отсутствовать. Приведем пример:

```
func_ref = !(x y) = (x*x - y*y);
```

```
set:(func_ref, ! (x y) [2] = sub:(mul:(x, x), mul:(y, y)))
=> ! (x y) [2] = sub:(mul:(x, x), mul:(y, y));
```

Здесь значением переменной *func_ref* станет ссылка на анонимный функтор (возвращающий разность квадратов двух своих аргументов). С этим значением допускаются те же операции, что и с любой ссылкой на функтор — прежде всего, косвенный вызов через **apply**:

```
func_ref ! (20, 10);
func_ref ! (5, 4);
```

```
apply:(func_ref, 20, 10) => 300;
apply:(func_ref, 5, 4) => 9;
```

Еще раз подчеркнем, что имя переменной *func_ref* принадлежит пространству имен *переменных*, а не функторов! И содержимым этой переменной в любой момент может стать ссылка на другой функтор (как и на значение какого-либо еще типа). Поэтому, если область действия функторов с именами определяется местом их описания (глобальные функторы действуют от своего описания до конца программы, локальные — до конца определяющего их функтора), то анонимные функторы после создания существуют до тех пор, пока в

программе активна хотя бы одна ссылка на них. Если последняя ссылка исчезает, то анонимный функтор считается более ненужным, и уничтожается автоматически. Поэтому декларировать анонимный функтор, не сохранив где-либо результат, совершенно бессмысленно — он будет уничтожен сразу после создания.
В остальном же, именованные и анонимные функторы вполне взаимозаменяемы.

Проверка на функтор

Иногда, перед использованием ссылки на функтор — надо бы проверить, действительно ли она таковой является. Это делается следующим образом:

Проверка на функтор	
Нормальная форма	***Семантика***
`is_func``(Q)`	*Значение Q — ссылка на функтор?*

Этот предикат возвращает *истину* (1), если его результат его операнда является ссылкой на произвольный функтор — и *ложь* (0), если он является чем-либо другим.

Функциональные операции

Многие встроенные функторы требуют ожидают ссылки на другой функтор в качестве аргумента. В частности, это относится к ряду строковых и списковых операций (которые мы ранее не рассматривали в т.ч. и по этой причине).

Кодирование символов

Как мы уже говорили, строка состоит из символов, каждый из которых имеет числовой *код*. Диапазон допустимых кодов (т. е. разрядность символов) определяется *типом* строки:

- **Тип 0**: *8-битовые* коды (диапазон 0..255),
- **Тип 1**: *16-битовые* коды (диапазон 0..65 535),
- **Тип 2**: *32-битовые* коды (диапазон 0..4 294 967 295).

Обычно, предполагается что строки типа 0 содержат символы набора ASCII, строки типа 1 — базовый набор Unicode (*Unicode-16*), а строки типа 2 — полный набор Unicode без ограничений. Вместе с тем заметим, что выбор интерпретации для кодов символов в основном определяется программистом. В частности, строки могут рассматриваться просто как удобные контейнеры для произвольных двоичных данных (даже графических или аудио). Это вполне законно, т. к. строка может содержать символы с *любыми* кодами из своего диапазона. В отличие от C, символ с кодом 0 вполне допустим в 8-битовой строке (и в любой другой). Аналогично, 16-битовая строка может содержать символы в диапазоне xD800..xE000 (что, строго говоря, недопустимо в стандарте Unicode). Хотя некоторые функторы и операции налагают строгие ограничения — в основном, интерпретация строковых данных больше определяется программистом, чем языком.

Возможен доступ не только к строке в целом, но и к ее отдельным символам. Проще всего это реализуется функтором **s_ord**.

Код символа в строке	
Нормальная форма	***Семантика***
s_ord(S, I)	*Код символа строки S с индексом I.*

Результат вычисления этого функтора — код символа строки *S,* имеющего индекс *I*. Индекс для строки интерпретируется так же, как и для списка: индекс 0 соответствует *первому* символу строки, 1 — *второму* и т. п. Допустимы и отрицательные индексы (-1 соответствует *последнему* символу в строке, -2 — *предпоследнему* и т. п.) Если параметр *I* опущен, предполагается 0 — т. е. **s_ord**(*S*) возвращает код первого символа строки. Заметим, что результат — это число, а не строка! Далее, когда мы будем говорить о кодах символов, речь тоже будет идти о неотрицательных числах (если специально не обговорено другое).

Для получения кода литерального символа проще использовать префикс \c.

Строковые конструкторы

Иногда нужно создать строку из символов с определенными кодами. Для этого в языке есть несколько операций.

Конструктор строки из списка кодов	
Нормальная форма	***Семантика***
s_chars(Type, Codes)	*Строка типа Type из символов списка Codes.*

Эта операция является (в некотором смысле) обратной для **s_ord**.
Первый аргумент для **s_chars** — это тип создаваемой строки (0, 1 или 2). За ним следует список целых значений (произвольной длины), задающих коды символов. Результатом является строка, содержащая символы с перечисленными кодами в заданном порядке (если список кодов пуст, то строка-результат тоже будет пустой).

```
s_chars (0, \c'A', \c'E', \c'I', \c'O', \c'U');
```

```
s_chars:(0, 65, 69, 73, 79, 85) => "AEIOU";
```

Таким образом нетрудно создать строку, содержащую символы с произвольными кодами. Однако, нередко нужно создать строку с набором строго упорядоченных (по возрастанию или убыванию) символов. Это проще делать операцией **s_range**.

Конструктор строки из диапазона	
Нормальная форма	***Семантика***
s_range(Type, Order, Range)	*Строка типа Type из символов диапазона Range (в порядке Order).*

Первый аргумент *Type* задает тип создаваемой строки. Второй — порядок символов (возрастающий или убывающий). Третий — собственно, диапазон символов, обычно в форме *StartCode..EndCode*, где *StartCode* — код младшего символа диапазона, а *EndCode* — *на 1 больше*, чем код старшего символа диапазона. Таким образом, строка-результат содержит *EndCode* - *StartCode* символов (но будет пустой, если *StartCode* >= *EndCode*). Если значение параметра *Order* равно 0, то первым символом строки будет *StartCode*, а последним — *EndCode*-1 (если оно отлично от 0, то наоборот). Так можно создать строку, содержащую все символы от «A» до «Z» (по возрастанию или по убыванию):

```
s_range (0, 0, \c'A' .. \c'Z'+1);
```

```
s_range:(0, 0, 65, add:(90, 1)) =>
"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
s_range (0, 1, \c'A' .. \c'Z'+1);
```

```
s_range:(0, 1, 65, add:(90, 1)) =>
"ZYXWVUTSRQPONMLKJIHGFEDCBA";
```

Это работает правильно, т. к. заглавные буквы латинского алфавита (как в ASCII, так и в Unicode) упорядочены по возрастанию. (Это же справедливо и для строчных букв, и для десятичных цифр.)

Строковые итераторы

Для строк также определены и свои циклические операции, т. е. *итераторы*:

Строковые итераторы	
Нормальная форма	***Семантика***

Строковые итераторы	
s_loop(Var, String, Body)	*Выполнить Body для всех кодов символов строки String (в прямом порядке).*
s_loop_r(Var, String, Body)	*Выполнить Body для всех кодов символов строки String (в обратном порядке).*

Все эти итераторы работают подобно списковым итераторам (**l_loop** и **l_loop_r**) — но роль списка здесь играет строка. Оба итератора последовательно присваивают *коды* символов из строки *String* переменной *Var*, после чего выполняют тело цикла *Body*. При этом **s_loop** перебирает символы в прямом порядке (от первого к последнему), а **s_loop_r** — в обратном порядке (от последнего к первому). Результатом операций является результат последнего выполнения *Body* (или (), если строка *String* является пустой). Как и для всех итераторов, единственный способ досрочного выхода из **s_loop** — исключение или **return**.
Примерно таким образом можно вычислить контрольную сумму всех символов в строке *Str*:

```
Checksum = 0;
s_loop (code, Str, Checksum =+: code);
```

В реальных программах, конечно, обычно применяются менее тривиальные алгоритмы вычисления контрольных сумм — но общая идея та же.

Классы символов

Символы могут принадлежать определенным семантическим *классам* (таким, как цифровые, алфавитные, пунктуационные и пр.) (Некоторые символы принадлежат сразу нескольким классам.) Для классификации символов имеются следующие встроенные предикаты:

Предикаты для проверки классов символов:	
Нормальная форма	***Семантика***
cc_blank(Code)	*Code — пробельный символ?*
cc_lower(Code)	*Code — буква нижнего регистра?*
cc_upper(Code)	*Code — буква верхнего регистра?*
cc_alpha(Code)	*Code — буква (алфавитный символ)?*
cc_digit(Code)	*Code — десятичная цифра?*
cc_odigit(Code)	*Code — восьмеричная цифра?*
cc_xdigit(Code)	*Code — шестнадцатеричная цифра?*
cc_print(Code)	*Code — печатаемый символ?*

Для всех перечисленных предикатов операндом является *код символа* (целое число, не строка!), а результатом — 1 (если символ принадлежит к соответствующему классу) или 0 (если он ему не принадлежит). Например, **cc_upper**(\c ”a”) выдает 0, но **cc_alpha**(\c ”a”) выдает 1. (Префикс **cc_** — сокращение от *character class*.)
Хотя эти предикаты, разумеется, могут применяться и для проверки отдельных символов — они активнее используются совместно со строковыми операциями, описанными ниже.

Поиск символов в строке по критерию

Операции поиска подстроки в строке (**s_findfirst** / **s_findlast**) мы уже рассматривали выше. Следующие операции поиск позволяют найти в строке определенный *символ* (но не фиксированный, а заданный неким предикатом). Этих операций четыре: они различаются как по направлению поиска (с начала строки / с конца строки), так и по полярности проверки условия (истинное / ложное).

Подсчет символов по предикату	
Нормальная форма	***Семантика***
s_span_in(String, CCPred)	*Подсчет символов в начале строки String, для которых успешна проверка на CCPred.*
s_span_ex(String, CCPred)	*Подсчет символов в начале строки String, для которых неудачна проверка на CCPred.*
s_rspan_in(String, CCPred)	*Подсчет символов в конце строки String, для которых успешна проверка на CCPred.*
s_rspan_ex(String, CCPred)	*Подсчет символов в конце строки String, для которых неудачна проверка на CCPred.*

Эти операции отличаются от операций поиска подстроки в первую очередь тем, чем вторым операндом является не подстрока, а *ссылка на предикат*, задающий класс символов (например, любой из предикатов группы **cc_** из предыдущей главы). Функторы с суффиксом **_in** пропускают все символы, для которых вычисление *CCPred истинно* (другими словами — находят первый символ строки, для которого оно *ложно*). Для функторов с суффиксом **_ex** обратная полярность проверки: они пропускают все символы, для которых вычисление *CCPred ложно* (другими словами — находят первый символ строки, для которого оно *истинно*). Помимо этого, различается направление просмотра строки: функторы **s_span** просматривают ее *вперед* от *первого* символа, функторы **s_rspan** — *назад* от *последнего*. Независимо от направления просмотра — в качестве результата выводится количество пропущенных символов (с начала / с конца строки).

Например, вызов **s_span_in**(String, !**cc_digit**) возвращает число десятичных цифр в *начале* строки *String* (0, если их нет вообще; длину строки *String*, если она полностью состоит из десятичных цифр). Другой пример: вызов **s_rspan_ex**(String, !**cc_alpha**) подсчитывает все символы в *конце* строки *String*, которые *не являются* алфавитными (0, если последний символ строки является алфавитным; длину строки *String*, если в ней вообще нет алфавитных символов). Во всех случаях, минимальным возвращаемым значением является 0, а максимальным — длина строки *String*.

Эти операции очень полезны для быстрого лексического разбора строки, ее токенизации и т. п.

Фильтрация символов по критерию

Есть операции, позволяющие быстро отбросить все символы строки, *не удовлетворяющие* (или, наоборот, *удовлетворяющие*) какому-либо условию (также заданному предикатом).

Фильтрация символов по предикату	
Нормальная форма	***Семантика***
s_filter_in(String, CCPred)	*Фильтрация символов в строке String, для которых успешна проверка на CCPred.*
s_filter_ex(String, CCPred)	*Фильтрация символов в строке String, для которых неудачна проверка на CCPred.*

Как и выше, первым операндом является строка, вторым — *ссылка на предикат*. Результатом вычисления является строка, включающая *только* те символы строки *String*, для которых вычисление *CCPred истинно* (для **_in**) или *ложно* (для **_ex**). (Все прочие символы просто выбрасываются.)
Например, вызов **s_filter_ex**(String, !**cc_blank**) — возвращает строку *String*, из которой выброшены все пробельные символы. Напротив, вызов **s_filter_in**(String, !**cc_digit**) — возвращает строку *String*, в которой *оставлены только* десятичные цифры (т. е. выброшены все символы, *не* являющиеся цифрами).
Когда все символы удовлетворяют критерию (для **_in**) или не удовлетворяют (для **_ex**) — результатом просто является строка *String* без изменений. Напротив: если все символы *не* удовлетворяют критерию (для **_in**) или удовлетворяют (для **_ex**) — результатом будет пустая строка.

Подсчет символов по критерию

Эти операции позволяют подсчитать символы (не конкретно в начале или конце, а во *всей* строке), *не удовлетворяющие* (или, наоборот, *удовлетворяющие*) какому-либо условию (заданному, опять же, предикатом).

Фильтрация символов по предикату	
Нормальная форма	***Семантика***
s_count_in(String, CCPred)	*Подсчет символов в строке String, для которых успешна проверка на CCPred.*
s_count_ex(String, CCPred)	*Подсчет символов в строке String, для которых неудачна проверка на CCPred.*

Здесь тоже первым операндом является строка, а вторым — *ссылка на предикат*. Результатом вычисления является количество символов строки *String*, для которых вычисление *CCPred* будет *истинно* (для **_in**) или *ложно* (для **_ex**).
Очевидно что с помощью **s_count** можно сразу вычислить длину строки, возвращаемую **s_filter**:

s_count_in (String, CCPred) == **s_len** (**s_filter_in**(String, CCPred))
s_count_ex (String, CCPred) == **s_len** (**s_filter_ex**(String, CCPred))

Эти два тождества всегда справедливы.

Определяемые классы символов

Во всех примерах выше, мы использовали встроенные в язык предикаты (вроде **cc_digit** или **cc_alpha**) для проверки принадлежности символа к некоторой категории. Однако, нетрудно

определить и свои собственные. Один из простых способов сделать это — использовать специальные функторы-конструкторы:

Конструкторы для символьных предикатов	
Нормальная форма	***Семантика***
cc_incl(String)	*Предикат, истинный для всех символов в String (и ложный для остальных)*
cc_excl(String)	*Предикат, ложный для всех символов в String (и истинный для остальных)*

Необычность этих функторов (ранее мы подобные не рассматривали) состоит в том, что их результат также является анонимным функтором-предикатом (точнее говоря, ссылкой на него)! Предикат, создаваемый обращением к **cc_incl**, возвращает истину (1) для *кодов* всех символов из строки-операнда *String*, и ложь (0) для кодов всех остальных символов. Предикат, создаваемый обращением к **cc_excl**, работает точно наоборот: он возвращает ложь (0) для *кодов* всех символов из строки-операнда *String*, и истину (1) для кодов всех остальных символов. Результат этой операции является анонимным функтором, поэтому вызвать его по имени нельзя — но можно сделать это через **apply**:

```
cc_incl ("ABCDEF") ! \c"A"
```

— возвращает 1 (символ «A» есть в строке «ABCDEF»);

```
cc_incl ("ABCDEF") ! \c"I"
```

— возвращает 0 (символ «I» в этой строке отсутствует).
Хотя создаваемый этими операциями анонимный функтор можно вызвать и непосредственно — удобнее всего использовать его совместно с функторами из семейств **s_span**, **s_filter** или **s_count**. Фактически, он позволяет определить для этих функторов новый произвольный класс символов (путем их простого перечисления). Например, в результате выполнения:

```
cc_vowel = cc_incl("AEIOUYaeiouy");
```

переменная *cc_vowel* получит своим значением предикат, истинный для всех гласных букв латинского алфавита (и ложный для всех прочих символов). Использовать его можно, например, так:

```
s_filter_ex (String, cc_vowel);
```

— возвращает строку из всех символов *String*, которые *не являются* гласными;

```
s_span_in (String, cc_vowel);
```

— подсчитывает, сколько гласных есть в начале *String*.
Заметим, что символы в строке *String* могут повторяться (хотя практического смысла в этом и нет). Строка *String* может быть пуста: **cc_incl** (“”) возвращает *всегда ложный* предикат (синоним !**false**), а **cc_excl** (“”) возвращает *всегда истинный* предикат (синоним !**true**).

Замена символов

Для того, чтобы преобразовать строку путем последовательной замены символов (по фиксированному алгоритму) — можно использовать функтор **s_map**:

Преобразование символов в строке	
Нормальная форма	***Семантика***
s_map(Mapper, String)	*Строка, полученная последовательным вызовом Mapper для каждого символа строки String*

Первый операнд — *ссылка на функтор*. Единственным аргументом при его вызове является *число* (код символа, к которому он применяется), а его результатом также должно быть *число* (символ, в который он преобразуется). Работа **s_map** состоит в том, что для кода каждого символа в строке *String* применяется *Mapper*, и из выданных им кодов составляется результат. Заметим, что длина строки результата всегда совпадает с длиной строки-операнда (если операнд пуст — результат тоже).

Например, с помощью **s_map** очень легко определить алгоритм кодирования латинского текста методов «поворота на 13 символов» (он же Rot13). (Разумеется, этот наивный алгоритм «шифрования» предназначен не для реального сохранения тайны, а в лучшем случае, для защиты текста от случайного прочтения.) Метод кодирования состоит в том, что из 26 символов латинского алфавита — первые 13 заменяются на последние 13, и наоборот («**A**» ↔ «**N**», «**B**» ↔ «**O**», … «**L**» ↔ «**Y**», «**M**» ↔ «**Z**»). (Символы, не входящие в латинский алфавит — остаются без изменения.) Это легко реализовать так:

```
! rot_13 (string) =
    s_map (!(code) =
    (cc_lower(code) ? (code >= \c'n' ? code-13 : code+13)
     :
     cc_upper(code) ? (code >= \c'N' ? code-13 : code+13)
     :
     code), string);
```

Заметим, что **rot13** является одновременно алгоритмом и кодирования, и декодирования (т. к. его повторное применение уничтожает результаты первого). Он работает примерно так:

```
rot_13("Secret message here!");
  rot_13:"Secret message here!" => "Frperg zrffntr urer!";

rot_13("Frperg zrffntr urer!");
  rot_13:"Frperg zrffntr urer!" => "Secret message here!";
```

Смена регистра символов

Для изменения регистра символов в строке (*заглавный* ↔ *строчный*) — также, в принципе, можно использовать **s_map**. Однако, имеются и более простые (и удобные) способы.

Преобразование символов в строке	
Нормальная форма	***Семантика***
s_ucase(String)	*Перевести строку String в верхний регистр*
s_lcase(String)	*Перевести строку String в нижний регистр*
s_icase(String)	*Инвертировать регистр символов строки String*

Во всех этих функторах исходная строка *String* не меняется — но возвращается соответствующий результат. Для **s_ucase** или **s_lcase** — возвращается строка, переведенная (соответственно) в верхний или нижний регистр. Эффект от **s_icase** заключается в *инвертировании* регистра (т. е. замене всех заглавных букв на строчные, и наоборот) — при этом не алфавитные символы остаются без изменения.
Эти функторы применяются ко всем символам строки. При необходимости, можно изменить только первый ее символ:

Преобразование первого символа в строке	
Нормальная форма	***Семантика***
s_ucfirst(String)	*Перевести первый символ String в верхний регистр*
s_lcfirst(String)	*Перевести первый символ String в нижний регистр*
s_icfirst(String)	*Инвертировать регистр первого символа строки String*

Эти функторы работают совершенно аналогично трем предыдущим — за исключением того, что соответствующему изменению подвергается только первый символ строки.

Строково-числовые преобразования

Простейший способ преобразования числовых значений в строки состоит в использовании функтора **string**, а строковые значения преобразуются в числа, соответственно, с помощью **int** или **float**. Однако, довольно часто требуются более гибкие методы преобразований.
Для начала, рассмотрим преобразования между строками и целыми.

Преобразование строк в целые	
Нормальная форма	***Семантика***
n_dec(String)	*Преобразование строки String в десятичное целое*
n_hex(String)	*Преобразование строки String в шестнадцатеричное целое*
n_oct(String)	*Преобразование строки String в восьмеричное целое*
n_bin(String)	*Преобразование строки String в двоичное целое*
n_base(Base, String)	*Преобразование строки String в целое по основанию Base*

Самый универсальный функтор **n__base** преобразует строку *String* в число, записанное в системе счисления с основанием *Base*. (Минимальное значение Base равно 2, максимальное — 36.) В качестве цифр используются сперва обычные десятичные цифры, а

потом (когда *Base* > 10) — латинские буквы от A до Z (без учета регистра), для обозначения цифр от 10 до *Base*-1. Перед записью числа может быть любая последовательность пробельных символов (она игнорируется), а затем знак «+» или «-» (он учитывается). Запись числа завершается первым же символом, который не является законной «цифрой» для основания *Base*. Если ни одной цифры прочитать не удалось — результатом будет 0.
Прочие функторы являются просто сокращениями для самых популярных значений *Base*:

- **n_dec** (*String*) == **n__base** (10, *String*)
- **n_oct** (*String*) == **n__base** (8, *String*)
- **n_hex** (*String*) == **n__base** (16, *String*)
- **n_bin** (*String*) == **n__base** (2, *String*)

Следующие функторы выполняют обратные преобразования — из строк в числа:

Преобразование целых в строки	
Нормальная форма	***Семантика***
s_dec(Int)	*Преобразование целого Int в десятичную запись*
s_hex(Int)	*Преобразование целого Int в шестнадцатеричную запись*
s_oct(Int)	*Преобразование целого Int в восьмеричную запись*
s_bin(Int)	*Преобразование целого Int в двоичную запись*
s_base(Base, Int)	*Преобразование целого Int в запись по основанию Base*

Здесь самым универсальным является функтор **s_base**, преобразующий целое число *Int* в его запись по основанию *Base*. (Как и выше, *Base* может меняться от 2 до 36 включительно, и для цифр выше 10 используются латинские буквы.) Заметим, что преобразуемое число всегда рассматривается как *беззнаковое*: ко всем отрицательным числам прибавляется 1<<32. Соответственно, знак для числа никогда не генерируется.

Прочие функторы являются сокращениями для основных значений *Base*:

- **s_dec** (*Int*) ==$ **s__base** (10, *Int*)
- **s_oct** (*Int*) ==$ **s__base** (8, *Int*)
- **s_hex** (*Int*) ==$ **s__base** (16, *Int*)
- **s_bin** (*Int*) ==$ **s__base** (2, *Int*)

Наконец, рассмотрим и преобразования вещественных чисел в строки:

Преобразование вещественных чисел в строки	
Нормальная форма	***Семантика***
s_ffloat(Val, N)	*Преобразование числа Val в формат F (с точностью N)*
s_efloat(Val, N)	*Преобразование числа Val в формат E (с точностью N)*
s_gfloat(Val, N)	*Преобразование числа Val в формат G (с точностью N)*

Все эти операции преобразуют свой первый операнд *Val* в строку, содержащую десятичную запись числа в форматах F, E или G. Интерпретация второго операнда (точности) зависит от выбранного формата. Для формата F параметр *N* задает число выводимых цифр после десятичной точки. Формат E — экспоненциальный (число представляется в виде

мантисса+характеристика), и для него «точность» определяет число знаковых цифр. Формат G — определяет более подходящий из двух форматов, F или E.
Независимо от выбранного формата, отрицательное число выводится со знаком, а неопределенность («не число»), положительная и отрицательная бесконечность выводятся в виде «#NaN», «+#Inf» и «-#Inf».

Отображение списка на список

Рассмотрим также несколько функторов, применяющих к спискам ссылки на функторы.

Отображение списка	
Нормальная форма	***Семантика***
l_map(Func, List)	*Возвращает результат, полученный поэлементным применением Func к List*

Встроенный функтор **l_map** выполняет операцию “отображения” списка-операнда, определяемого функтором-операндом. Результатом его вычисления является *новый список* (сам *List* не изменяется!), состоящих из результатов применения *Func* к элементам *List*. Например:

```
l_map (!sin, (0, pi(1/3), pi(1/2), pi(1)));
```

```
l_map:(sin:, 0, pi:div:(1, 3), pi:div:(1, 2), pi:1) => (0.,
0.8660254, 1., 1.2246064e-016);
```

```
l_map (!(x) = (3*x + 2), [7 2 6 5 9]);
```

```
l_map:(! (x) [1] = add:(mul:(3, x), 2), 7, 2, 6, 5, 9) =>
(23, 8, 20, 17, 29);
```

```
l_map (!(str) = ('<' +$ str +$ '>'), [10 'aa' 20 'bb' 30
'cc']);
```

```
l_map:(! (str) [1] = s_cat:(s_cat:("<", str), ">"), 10,
"aa", 20, "bb", 30, "cc") => ("<10>", "<aa>", "<20>",
"<bb>", "<30>", "<cc>");
```

Если в качестве *List* задан скаляр (в общем случае, произвольное атомарное выражение), то результатом будет просто **apply**(*Func*, *List*). Если значение *List* пусто — то и результатом будет (). Заметим также, что если *List* является закрытым списком, то и результат будет закрытым; а если открытым — результат будет открытым.

Преобразование диапазона в список

Мы уже говорили, что целочисленный диапазон задается списком из его нижней и верхней границ (или же единственным числом, задающим верхнюю границу — в этом случае нижняя принимается равной нулю). При необходимости, диапазон легко «развернуть» в список, последовательно содержащий все входящие в него значения.

Развернуть диапазон в список	
Нормальная форма	***Семантика***

Развернуть диапазон в список	
l_range(Range, [Func])	*Преобразовывает диапазон Range в список из всех значений (по возрастанию), с применением к ним Func*
l_range_r(Range, [Func])	*Преобразовывает диапазон Range в список из всех значений (по убыванию), с применением к ним Func*

Оба функтора фактически делают одно и то же: преобразуют диапазон *Range* в список всех значений в этом диапазоне. Разница в том, что результатом **l_range** всегда является *возрастающий* список (от нижней границы — до верхней минус один), а результатом **l_range_r** — *убывающий* список (от верхней границы минус один — до нижней). Если диапазон пуст — то и результатом будет ().
Второй операнд (*Func*) является необязательным. Если он присутствует, то он задает ссылку на функтор, применяемый к каждому элементу результата. Т.о. фактически **l_range** (*Range*, *Func*) равносильно **l_map** (*Func*, **l_range** (*Range*,)) — но первая форма намного эффективнее. Заметим еще, что даже когда *Func* отсутствует, запятая после *Range должна* присутствовать (т. к. без нее список-диапазон «распадется» на свои составляющие).

Поиск элемента в списке

В языке есть набор функторов, обеспечивающих *поиск элемента в списке* по заданному критерию. Система операций поиска имеет двойную симметрию: они различаются как направлением поиска (*вперед с начала* / *назад с конца*), так и полярностью проверки условия (*истинное* / *ложное*).

Поиск в списке по предикату	
Нормальная форма	***Семантика***
l_while(Pred, List)	*Найти первый (с начала) элемент List, для которого неудачна проверка на Pred.*
l_while_r(Pred, List)	*Найти последний (с конца) элемент List, для которого успешна проверка на Pred.*
l_until(Pred, List)	*Найти первый (с начала) элемент List, для которого успешна проверка на Pred.*
l_until_r(Pred, List)	*Найти последний (с конца) элемент List, для которого неудачна проверка на Pred.*

Функтор **l_while** просматривает список *List* (вперед с начала), применяя к каждому элементу предикат *Pred*. Возвращается индекс первого элемента, для которого тест *неудачен* (0, если он неудачен для начального элемента); если же тест был удачен для всех элементов — возвращается *#List* (длина списка). Функтор **l_while_r** просматривает список в обратном направлении, и возвращает *на 1 больше*, чем индекс первого (с конца) элемента, для которого тест *неудачен* (т. е. индекс последнего с конца элемента, для которого тест *удачен*). Как крайние случаи, возвращается *#List* (если тест неудачен для последнего элемента) или 0 (если тест удачен для всех элементов). Таким образом, оба функтора возвращают результат в диапазоне 0..*#List*.
Функторы **l_until** и **l_until_r** отличаются от **l_while** и **l_while_r** лишь тем, что полярность проверки условия у них противоположная.

Вот примеры применения всех этих функторов (как и их результаты):

```
List = [10 20 30 40 50];

for_inc (i, 6, {
     j = 5 + 10*i;
     <: ('l_while ( <', j, ' ) = ', l_while (!(x) = (x < j),
list), '\t');
     <: ('l_while_r ( >', j, ' ) = ', l_while_r (!(y) = (y >
j), list), '\n');
     <: ('l_until ( >', j, ' ) = ', l_until (!(x) = (x > j),
list), '\t');
     <: ('l_until_r ( <', j, ' ) = ', l_until_r (!(y) = (y <
j), list), '\n');
     });
```

```
l_while ( <5 ) = 0       l_while_r ( >5 ) = 0
l_until ( >5 ) = 0       l_until_r ( <5 ) = 0
l_while ( <15 ) = 1      l_while_r ( >15 ) = 1
l_until ( >15 ) = 1      l_until_r ( <15 ) = 1
l_while ( <25 ) = 2      l_while_r ( >25 ) = 2
l_until ( >25 ) = 2      l_until_r ( <25 ) = 2
l_while ( <35 ) = 3      l_while_r ( >35 ) = 3
l_until ( >35 ) = 3      l_until_r ( <35 ) = 3
l_while ( <45 ) = 4      l_while_r ( >45 ) = 4
l_until ( >45 ) = 4      l_until_r ( <45 ) = 4
l_while ( <55 ) = 5      l_while_r ( >55 ) = 5
l_until ( >55 ) = 5      l_until_r ( <55 ) = 5
```

Фильтрация списка

Следующие два функтора (различающиеся лишь полярностью условия) позволяют выбрать из списка только те элементы, которые отвечают заданному критерию.

Фильтрация списка по предикату	
Нормальная форма	***Семантика***
l_filter_in(Pred, List)	*Список всех элементов List,* *для которого успешна проверка на Pred.*
l_filter_ex(Pred, List)	*Список всех элементов List,* *для которого неудачна проверка на Pred.*

Оба функтора возвращают списки, состоящие из всех элементов списка *List*, для которых тест (т. е. применение *Pred*) *удачен* (**l_filter_in**) или *неудачен* (**l_filter_ex**). (Оригинал списка, конечно, не меняется.) Если тест *удачен* для всех элементов списка, **l_filter_in** возвращает сам список, а **l_filter_ex** возвращает (). Напротив, если он *неудачен* для все элементов — **l_filter_ex** возвращает сам список, а **l_filter_in** возвращает ().
Примеры:

```
List = [10 20 30 40 50];
l_filter_in (!(x) = ((x % 10) & 1), List);
l_filter_ex (!(x) = ((x % 10) & 1), List);
```

```
l_filter_in:(! (x) [1] = and:(idiv:(x, 10), 1), list) =>
(10, 30, 50);
l_filter_ex:(! (x) [1] = and:(idiv:(x, 10), 1), list) =>
(20, 40);
```

Подсчет элементов в списке

Следующие функторы подсчитывают все элементы списка, для которых тест успешен или неудачен:

Подсчет элементов по предикату	
Нормальная форма	***Семантика***
l_count_in(Pred, List)	*Количество элементов List, для которых успешна проверка на Pred.*
l_count_ex(Pred, List)	*Количество элементов List, для которых неудачна проверка на Pred.*

Как и для **l_while**/**l_until**, результатом здесь является целое число — но подсчитываются не только начальные/конечные, но *все* элементы списка *List*, для которых тест с помощью предиката *Pred* успешен (**l_while**) или неудачен (**l_until**).
Понятно также, что результат этих операций совпадает с длиной результата **l_filter_in**/**l_filter_ex**.

Сравнение списков

Сравнение на упорядоченность (меньше / больше / равно) может быть определено не только для скаляров, но и для списков. Однако, для этого требуется дополнительный операнд-функтор, определяющий, как именно сравниваются его элементы:

Сравнение списков с помощью компаратора	
Нормальная форма	***Семантика***
l_cmp(Comp, ListL, ListR)	*Поэлементное сравнение списков ListL и ListR, применяя компаратор Comp.*

Функтор сравнения (*компаратор*) — это функтор, который применяется к каждой паре элементов из списков *ListL* и *ListR* для сравнения их на упорядоченность. В качестве результата — компаратор должен возвращать *отрицательное* значение (если первый операнд меньше второго), *положительное* значение (если первый операнд больше второго) или 0 (если операнды равны). Нам уже знакомы некоторые встроенные в язык компараторы: **cmp** (для сравнения числовых значений) и **s_cmp** (для словарного сравнения строк). Для менее тривиального упорядочения, можно определить свой собственный. Заметим, что стандартные компараторы всегда возвращают строго -1 (когда первый операнд «меньше» второго), или 1 (когда первый операнд «больше» второго). Для пользовательского компаратора требования не такие строгие (как результат, подойдет любое отрицательное или положительное значение), но (в любом случае) результат должен быть числом.

Сравнение списков работает примерно так же, как и словарное сравнение строк (с той очевидной разницей, что оно проходит поэлементно, а не посимвольно). А именно, к каждому элементу из *ListL* и элементу из *ListR* применяется компаратор *Comp* — до тех пор, пока его результатом является 0 (или, пока один из списков не кончится). Первый ненулевой результат компаратора является результатом **l_cmp**. Если один из списков короче другого (и при этом все его элементы компаратор считает «равными» элементам другого списка) — то результатом операции является -1 (если *ListL* короче *ListR*) или 1 (если *ListL* длиннее *ListR*). И только в том случае, когда длины списков равны, и все их элементы также «равны» (т. е. сравниваются в 0) — **l_cmp** возвращает 0.

Сортировка

В языке определены два разных функтора, связанных с сортировкой данных.

Сортировка	
Нормальная форма	***Семантика***
l_sort_index(Comp, Range)	*Возвращает список из диапазона Range, сортированный компаратором Comp.*
l_sort_mutator(Count, Acsr, Comp)	*Сортировка Count элементов, доступных через Acsr с помощью Comp*

Два этих функтора реализуют два разных подхода к сортировке.
Первый из них, собственно, сам ничего не сортирует — он лишь порождает список-индекс, отсортированный в соответствии с критерием, определенным компаратором. Диапазон списка задается аргументом *Range* (т. е. список порождается так же, как и при вызове **l_range**(*Range*)) — но созданный список возвращается упорядоченным не по возрастанию/убыванию, а в соответствии с *Comp* (т. е. таким образом, что применение *Comp* к элементам является возрастающим).
Второй *реально* выполняет сортировку данных, которые предоставляются ему не прямо, а опосредованно (через ссылку на функтор-аксессор *Acsr*). Элементы, выданные *Acsr*, упорядочиваются физически, в соответствии с порядком, заданным *Comp*.

Вычисления по требованию

Одна из самых мощных возможностей языка — непосредственное управление порядком вычислений. Например, язык дает возможность *откладывать* вычисление (или выполнение) любого выражения до того момента, когда это станет необходимо. Такой механизм (называемый *ленивыми вычислениями*) присутствует во многих языках. Но (в отличие от некоторых языков, поддерживающих «ленивость» вычислений) в AWL они полностью *управляемые* — программист имеет полный контроль над тем, что и когда будет вычислено. Для управления вычислениями предусмотрено несколько операций, которые мы раньше не рассматривали.

Девальвация и ревальвация

Эти операции AWL не имеют никакого отношения к миру финансов — но вот к вычислениям (и управлению ими) имеют самое прямое. Хотя мы уже затрагивали тему управления вычислениями — до сих пор мы не обсуждали никакие связанные с этим средства.

Дело в том, что в большинстве случаев понятие *вычисления* неявно заложено в семантику соответствующего функтора: свои аргументы он вычисляет автоматически. Например, это относится ко всем скалярным операциям (обсуждая их операнды, мы фактически имели в виду *результаты вычисления* операндов). Если операндом является литеральное значение (число или строка) — оно «вычисляется» в себя; если переменная — результатом является ее содержимое; если нетривиальное выражение — оно автоматически вычисляется, и используется его результат. Обычно операция явного вычисления не требуется — но она становится очень актуальной, если автоматическое вычисление какого-то операнда ранее было *отложено*. Таким образом, *подавление* (или откладывание) вычисления (*девальвация*) и вычисление ранее отложенного (*ревальвация*) — это две взаимно дополнительных операции (т. к. вторая «уничтожает» результат первой).

Девальвация и ревальвация		
Синтаксис	***Нормальная форма***	***Семантика***
`@Expr`	**`deval`**`(Expr)`	*Девальвация: результат — само выражение Expr (вместо результата его вычисления)*
`^Expr`	**`reval`**`(Expr)`	*Ревальвация: результат — результат результата Expr*

Фактически, применение унарной операции «@» (*девальвации*) к любому операнду — защищает (или *экранирует*) его от вычисления (то есть, ее результатом является сам операнд, а не *его* результат). Унарная операция «^» (*ревальвация*) обычно устраняет результат предыдущей девальвации: она возвращает *результат результата* (т. е., фактически, вычисляет свой операнд *дважды*).

Обе операции могут быть вложенными на произвольную глубину. Как это происходит, лучше всего продемонстрировать на примере:

```
V = @@@(2*2);
```

```
set:(V, deval:deval:deval:mul:(2, 2)) => deval:deval:mul:
(2, 2);
```

```
V;
```
```
  V => deval:deval:mul:(2, 2);
```
```
^V;
```
```
  reval:V => deval:mul:(2, 2);
```
```
^^V;
```
```
  reval:reval:V => mul:(2, 2);
```
```
^^^V;
```
```
  reval:reval:reval:V => 4;
```

Если б мы непосредственно присвоили переменной *V* выражение 2*2 — очевидно, она получила бы значение 4. Но перед этим выражением стоит несколько операций @ — каждая из которых защищает его от одного шага вычислений. Каждая операция ^ отменяет одну из @ — суммарно, пришлось применить три ^, чтобы, наконец, получить в результате 4.
Зачем нужны отложенные вычисления? В данном случае, конечно, не особенно нужны, так как произведением 2 и 2 всегда является 4 (независимо от того, в какой точке программы было умножение). В принципе, это справедливо для всех скалярных операций: они являются *чистыми* (т. к. результат их зависит исключительно от значений операндов). Однако, помимо скалярных операций, в AWL есть и переменные (а также списки, и другие *мутабельные* объекты), результат которых меняется в процессе выполнения программы. Кроме того, есть операции *с побочным эффектом* (например, ввод и вывод), для которых точный момент их выполнения важен.

«Ленивое» присваивание

Кроме обычного присваивания (функтор **set**), в языке есть еще т. н. «ленивое»:

Ленивое присваивание		
Синтаксис	***Нормальная форма***	***Семантика***
`V := W`	**`let`**`(V, W)`	*«Ленивое» присваивание:* *присвоить V выражение W без его вычисления*

Операция *ленивого присваивания* (ее реализует встроенный функтор **let**) интерпретирует свои операнды так же, как и **set**: правый операнд может быть произвольным выражением, левый должен быть мутабельным. Принципиальная разница состоит в том, что правый операнд *не вычисляется*, а непосредственно присваивается левому. Эффект ленивого присваивания — тот же, что и у обычного с девальвацией второго операнда (т.е выражение *V* := *W* всегда равносильно *V* = @*W*). (Поэтому без отдельной операции для него можно было и обойтись — но одна операция эффективнее, и удобнее в применении, чем две.)
Например:

```
S := (a + b)*2;
T := (a*b, b*c, c*a);

S; T;
```
```
  S => mul:(add:(a, b), 2);
  T => (mul:(a, b), mul:(b, c), mul:(c, a));
```

Здесь значениями переменных *S* и *T* станут *сами выражения*, вместо результатов их вычислений. Совершенно неважно, какие значения имеют (и имеют ли вообще) переменные *a*, *b* и *c* в момент присваивания. Это существенно лишь тогда, когда произойдет вычисление значений *S* и *T* (обычно, путем их ревальвации). Например:

```
[a b c] = [2 3 5];

^S; ^T;
```

```
reval:S => 10;
reval:T => (6, 15, 10);
```

```
[a b c] = [6 10 13];

^S; ^T;
```

```
reval:S => 32;
reval:T => (60, 130, 78);
```

Ленивое присваивание является мощным механизмом, аналоги которого в процедурных языках программирования встречаются редко. Отчасти, можно рассматривать его как разновидность *макроподстановки* (однако — в отличие, например, от препроцессора в C и C++ — подстановка здесь не просто текстуальная, а *семантическая*). Можно также рассматривать "ленивые" выражения как упрощенный вид определяемых пользователем функций: без параметров и локальных переменных. (Хотя, заметим, что с чисто технической точки зрения, функторы и выражения с отложенным вычислением — это совершенно разные механизмы, и путать их не следует!)

«Ленивая» передача параметров

Мы уже видели, что в языке многие функторы (например, условные и циклические) явно управляют вычислением своих аргументов. Для пользовательских функторов, параметры по умолчанию передаются с автоматическим вычислением — но его можно отключить. Для этого достаточно в декларации функтора поставить перед параметром символ «@». Эффект этого аналогичен тому, как если б к соответствующему аргументу функтора при вызове всегда применялась операция «@». То есть, аргумент передается функтору без вычисления — и ответственность за него аргумента перекладывается на сам функтор.
Используя «ленивую» передачу параметров, достаточно нетрудно определить новые условные, циклические и многие иные операции. Приведем лишь несколько примеров.
Как мы видели выше, большинство условных функторов языка существует в двух зеркально-симметричных разновидностях (различающихся лишь полярностью условия). В принципе, можно было бы обойтись лишь одной из них (тем более, что альтернативную тогда легко будет определить). Например, нетрудно определить условный функтор **unless** через функтор **if** (и наоборот — **if** через **unless**): и то, и другое делается буквально в одну строку:

```
! unless(cond @else @then) = if(~~cond, ^then, ^else);

! if(cond @then @else) = unless(~~cond, ^else, ^then);
```

Обе ветви условия (*then* и *else*) здесь передаются «лениво» (перед параметрами стоит префикс «@»), т.е. для их вычисления в теле функтора нужна ревальвация («^»). К условию *cond* «ленивая» передача не применяется, т. к. здесь в ней нет нужды (почему?) — поэтому и

ревальвация перед обращением к нему не нужна.
Так что, система из двух условных функторов несколько избыточна, и можно было б ограничиться любым из них — оба встроены в язык в основном потому, что так система условных операций приобретает красивую симметрию. Столь же избыточна и система циклических операций, т.к. **while** можно определить через **until** (а **until** — через **while**):

```
! while(@cond @loop) = until(~~^cond, ^loop);

! until(@cond @loop) = while(~~^cond, ^loop);
```

Вот здесь условие цикла *cond* уже *необходимо* передавать «лениво» (почему?) — соответственно, в теле функтора перед проверкой применяется ревальвация. Что же касается версий цикла с *постусловием* — то их тоже можно определить через версии с предусловием:

```
! do_while(@cond @loop) = { ^loop; while(~~^cond, ^loop) };

! do_until(@cond @loop) = { ^loop; until(~~^cond, ^loop) };
```

Эти определения не абсолютно идентичны встроенным в язык (т. к. встроенные всегда возвращают в качестве значения результат последней итерации *body* — но здесь, если последняя итерация была и первой, этот результат теряется). Впрочем, для большинства практических применений этим можно пренебречь. Как видим, хотя в языке есть *четыре* вида циклов с условием — вполне можно было обойтись любым из них (а остальные, при желании, определить через него).
Впрочем, определять уже реализованные в языке функторы — занятие довольно бессмысленное! Интереснее будет определить что-то такое, чего в языке еще нет. Например, в AWL отсутствует точный аналог C или Java-подобного оператора **for**. Однако, если функтор-итератор с очень похожей функциональностью легко определить в одну строку:

```
! c_for (@Init @Cond @Iter @Body) =
{^Init; ^Cond ?? {^Body; ^Iter}};
```

Использовать его так же просто, как **for** в C (только синтаксис вызова будет как и у всех функторов):

```
c_for (I = 1, I <= 5, ++ I, <: ['I = ' I '\n']);
```

```
I = 1
I = 2
I = 3
I = 4
I = 5
c_for:(set:(I, 1), le:(I, 5), inc:I, f_put:((), "I = ", I,
"
")) => 6;
```

Приведем другой пример: «арифметический **if**», выбирающий для выполнения одну из трех ветвей условия, в зависимости от того, положителен, отрицателен или равен нулю первый операнд. (Это похоже на арифметический **if** в Фортране — только работает, разумеется, не с метками и переходами, которых нет в AWL, а с ветвями условий.)

```
! aif(val @IfNeg @IfZero @IfPos) =
  val > 0 ? ^IfPos : val < 0 ? ^IfNeg : ^IfZero;
```

Как и для обычный условных функторов, **aif** одновременно и выбирает ветвь выполнения — и возвращает вычисленное ею значение:

```
<: («Число », a, «: », aif(a, «отрицательно», «равно нулю»,
«положительно»);
```

Наконец, приведем еще один любопытный пример. В некоторых языках (например, в Python) циклы с предусловием могут также иметь ветвь **else** (которая выполняется только в тех случаях, когда сам цикл не был выполнен ни разу). В AWL такого цикла нет — но определить его совсем нетрудно:

```
! while_else (@Cond @Loop @Else) =
unless (while (^Cond, {^Loop; 1})):: (^Else);
```

Здесь мы использовали небольшой трюк: телом цикла **while** мы сделали *блок*, который всегда возвращает значение 1. Поскольку результат **while** — это всегда результат последнего выполнения его тела, то им станет 1 (если цикл выполнился хоть раз) или (), если он вообще не выполнялся. Этот результат передается как условие **unless** — и если он будет пуст (т. е. цикл не выполнялся ни разу), то будет выполнено *Else* (иначе оно игнорируется).
Для все той же симметрии, аналогично определим цикл **until_then** (с обратной полярностью условия):

```
! until_then (@Cond @Loop @Then) =
unless (until (^Cond, {^Loop; 1})):: (^Then);
```

Наконец, отметим, что «ленивая» передача параметров может использоваться для имитации их передачи *по ссылке*. Многие сложные структуры данных (например, списки) всегда передаются по ссылке — но вот для передачи *изменяемых скаляров* требуются дополнительные трюки. Примерно так можно реализовать *циклические* инкремент и декремент:

```
! cycle_inc(@var N) = (++ ^var == N ? (^var = 0) : );

! cycle_dec(@var N) =
{ ^var == 0 ? (^var = N) : ; -- (^var); };
```

Операции циклически увеличивают и уменьшают свой первый операнд *var* (естественно, передаваемый «лениво») по модулю *N* (т. е. так, что он всегда остается в диапазоне 0..N). (Для простоты, проверка того, находится ли он в диапазоне 0..N изначально не производится.)

Функторы-оболочки

Помимо условий и итераторов, имеется еще один весьма важный класс функторов, которые мы ранее не рассматривали: *оболочки* (wrappers). Функторы эти также имеют один (как минимум) «лениво» передаваемый параметр. Но, если условные функторы обеспечивают необязательность выполнения своего параметра, а циклические — ее многократность, то оболочки выполняют его один (и только один) раз. Однако, при этом, перед выполнением своего тела оболочки выполняют некоторые предварительные действия (*пролог*), а после

него — завершающие действия (*эпилог*). Смысл применения оболочек заключается в том, что если был выполнен пролог, то и эпилог также непременно будет выполнен (даже если выполнение тела оболочки было завершено аномально — например, через исключение). Таким образом, оболочки эффективно обеспечивают целостность неких данных.
Приведем пример простой оболочки, упрощающей генерацию HTML (или XML) кода:

```
! HTag (name @content) =
    #content ? {
         <: ('<', name, '>');
         ^content;
         <: ('</', name, '>');
         }
    :
         (<: ('<', name, ' />'));
```

Для нового функтора **HTag** первым операндом является *имя тэга*, а вторым — его содержимое (которым может быть, собственно, любой код). Все, что делает **HTag** — обеспечивает выполнение своего второго аргумента, но перед ним автоматически выводит открывающий тэг, а после него — закрывающий. Если содержимое *content* пусто, генерируется пустой (не имеющий вложенного контента) тэг, в соответствии с форматом XML (и XHTML). Заметим еще, что для вызова **HTag** (name, content) обычно удобна и наглядна нотация **HTag** (name):: content.
HTag удобен для генерации вложенного контента, но у него есть особенность: вторым операндом для него всегда является некое выполняемое *действие*. Т.е. **HTag** сам не умеет выводить, например, скалярные операнды: для их вывода желательно определить примерно следующее:

```
! _(operand) = <: (operand);
```

Этот функтор настолько тривиален, что мы не будем придумывать ему содержательное имя — символа подчеркивания достаточно. Эти два функтора можно использовать совместно, например, так:

```
! HTML_format (title @body) =
    HTag 'HTML':: {
         HTag 'HEAD':: HTag 'TITLE':: _(title);
         HTag 'BODY':: (^body);
         };
```

То, что мы сейчас определили — довольно простая оболочка-шаблон для форматирования стандартного HTML документа. Новый функтор имеет только два параметра-атрибута: заголовок *title* и (собственно) контент *body*. Любое обращение к **HTML_format** вызовет вывод готового HTML-кода, где содержимое документа *body* (оно целиком на ответственности пользователя) автоматически заворачивается в стандартную HTML-оболочку (с тэгами **HTML**, **HEAD** и **BODY**, и с заголовком). Для более сложных документов, конечно, потребуется куда более сложный шаблон — но общая идея определения таких шаблонов должна быть понятно. Основное преимущество здесь заключается не только в более компактной записи — но и в том, что (в отличие от статичного HTML или XML) шаблоны имеют динамическую природу: они совмещают генерацию кода с произвольными вычислениями, позволяя создавать сложные Web-

страницы (или XML-документы) «на ходу».
Функторы-оболочки — довольно мощное средство для обеспечения целостности внутренних, недоступных программисту, данных. Довольно многие библиотечные компоненты AWL основаны на использовании оболочек (в т.ч. связанные с графикой и аудио).

Зацикливание ссылок

Как и любым мощным средством, отложенными вычислениями легко можно злоупотребить! Например, существует опасность *зацикливания ссылок* (в некотором смысле, аналогичного рекурсивному зацикливанию для функторов).
Достаточно написать что-нибудь вроде:

```
a := ^a + 1
```

После этого попытка вычислить значение ^a приведет к бесконечному рекурсивному циклу (который, впрочем, быстро завершится крахом интерпретатора).
Мораль здесь одна: при использовании «ленивых» вычислений, желательно соблюдать разумную осторожность.

Массивы

Создание массива

Массивы в AWL — это альтернатива спискам. *Массив* — это структурный тип данных, содержащий упорядоченный набор значений, каждое из которых доступно по целочисленному *индексу* (или по списку индексов, так как массив в общем может быть многомерным). Как и у списков, все элементы массива *мутабельны*. В отличие от списков, размеры массива задаются при его создании. (Их изменение после создания массива возможно — но требует определенных усилий.)

Имена функторов, связанных с массивами, по соглашению начинаются с **a_**. Новый массив создается вызовом примитивного конструктора **array** (для него есть синоним **a_create**):

Конструктор массива	
Нормальная форма	***Семантика***
array(Dim1, Dim2, … DimN)	*Создает новый массив (с размерами Dim1, Dim2 ... DimN)*

Операндом для **array** является *список размеров* массива (который вырождается в скаляр, когда массив одномерен). (Далее, мы будем называть этот список *метрикой* массива.) Любой из размеров массива может быть задан произвольным выражением (которое вычисляется, как целое). Если массив имеет более одного размера, то первый элемент списка мы будем называть *внешним* размером массива, а последний — *внутренним*. Общее количество измерений массива мы будем называть его *рангом*: например, ранг линейного массива — 1, а ранг прямоугольной матрицы — 2. Максимальный допустимый ранг массива — 16 измерений.

```
` создать вектор из 50 элементов `
Vector1 = array (50);

` создать матрицу из 40 строк и 70 столбцов `
Matrix2 = array (40, 70);
```

Ни один из размеров массива не должен быть отрицательным. Но любой размер массива может быть равен 1. Он даже может быть равен 0, но в этом случае массив не будет содержать ни единого элемента (хотя полезность такого массива сомнительна, как частный случай это допускается). Также заметим, что (в отличие от списков!) массив с размерами 0 или 1 не “вырождается” в примитивные элементы данных (т. е. в скаляр или в ()). Он всегда остается массивом, с той структурой, с котором создавался.

Массивы в AWL являются *гетерогенными*, т. е. могут содержать данные любого типа (и разные элементы могут иметь различные типы). Для только что созданного массива все его элементы *пусты*, т. е. инициализированы в ().

Интерпретатор выводит массив в виде списка его размеров (в том порядке, в каком они перечислены в конструкторе), заключенных в квадратные скобки и разделенных пробелами. Например, для созданных выше массивов:

```
Vector1 => [50];
Matrix2 => [40 70];
```

(Заметим, что значения самих элементов массива — по умолчанию не выводятся. Выводится только его метрика.)

Проверка на массив

Для нее есть следующий функтор-предикат:

Проверка на массив	
Нормальная форма	***Семантика***
is_array(Value)	*Истинно, если Value является массивом*

Выражение **is_array**(V) возвращает истинное значение, если результат *V* — произвольный массив (и ложное в противном случае).

Доступ к элементам массива

Чаще всего работа с массивом выполняется поэлементно, для чего предназначен встроенный аксессор **a_elem**.

Доступ к элементам массива		
Синтаксис	***Нормальная форма***	***Семантика***
Array{Inx1, …, InxN}	**a_elem**(Array, Inx1, …, InxN)	*Доступ к элементу Array (с индексами Inx1 … InxN)*

Число индексов *N* должно быть равно рангу массива *Array*. Порядок индексов соответствует порядку размеров массива: *Inx1* соответствует самому внешнему измерению, *InxN* — самому внутреннему. Для каждого измерения индексы начинаются с нуля (т. е. если размер массива по соответствующему измерению равен *L*, то минимальный индекс равен 0, а максимальный — *L-1*).
Поскольку операция **a_elem** является *аксессором*, она предоставляет мутабельный доступ к элементу массива с соответствующими индексами. Например:

```
Vector1{25} = 22.33;
++ Matrix2{8, 12};
```

Если какой-либо из индексов находится вне диапазона — возвращается пустое значение (не мутабельное). Изменить размеры массива путем обращения к его несуществующим элементам — *невозможно*.

Метрика и ранг массива

Несколько примитивных операций позволяют получить информацию о ранге и о размерах массива.

Метрика и ранг массива	
Нормальная форма	***Семантика***
a_dims(Array)	*Метрика массива Array*
a_rank(Array)	*Ранг массива Array*
a_total(Array)	*Количество элементов в массиве Array*

Функтор **a_dims**(*Array*) возвращает список размеров (метрику) массива *Array* (в порядке конструктора, т. е. от внешних к внутренним).

```
a_dims (Vector1);
a_dims (Matrix2);
```

```
a_dims:Vector1 => 50;
a_dims:Matrix2 => (40, 70);
```

Функтор **a_rank**(*Array*) возвращает ранг массива *Array* (что всегда равно #**a_dims**(*Array*)). Например:

```
a_rank (Vector1);
a_rank (Matrix2);
```

```
a_rank:Vector1 => 1;
a_rank:Matrix2 => 2;
```

Наконец, результатом для **a_total**(*Array*) является общее количество элементов в массиве *Array*. (Оно всегда равно произведению всех его размеров, т. е. [=] * **a_dims**(*Array*))

```
a_total (Vector1);
a_total (Matrix2);
```

```
a_total:Vector1 => 50;
a_total:Matrix2 => 2800;
```

Заполнение массива

Если нужно присвоить всем элементам массива одно и то же значение, удобно использовать примитив **a_fill**(*Array*, *Value*). В результате, значение *Value* присваивается всем элементам массива *Array*.

Заполнение массива	
Нормальная форма	***Семантика***
a_fill(Array, Value)	*Присвоить Value всем элементам Array*

Например:

```
a_fill (Matrix2, 0);
```

— полностью обнуляет все элементы матрицы *Matrix2*.

```
a_fill (Vector1, "");
```

— присваивает всем элементам *Vector1* пустую строку.
Если элемент *Value* опущен, предполагается () — поэтому **a_fill**(*Array*) вызывает полную очистку массива *Array*, присваивая всем его элементам значение ().
Также заметим, что все элементы массива являются ссылками на *единственный экземпляр Value* — никакого копирования не происходит. Это не принципиально для скаляров, но надо иметь в виду, если *Value* является мутабельной структурой данных (например, списком).

Итератор по массиву

Практически для всех сложных структур данных в языке имеются свои итераторы — и массивы не исключение.

Итератор по массиву	
Нормальная форма	***Семантика***
a_loop(Array, Var, Loop)	*Выполнить Loop, для значений Var, равных всем элементам из массива Array.*

Итератор по массиву **a_loop** работает подобно всем прочим итераторам языка. Он (как и почти все прочие итераторы) тернарен: параметр *Array* задает массив, *Var* — переменную (или произвольное мутабельное), *Loop* — тело цикла. Выполнение **a_loop** состоит в том, что перебираются все элементы массива, значение каждого из них присваивается переменной *Var*, после чего выполняется тело итератора *Loop*. Порядок перебора элементов жестко фиксирован: по возрастанию индексов, и от внутренних индексов — к внешним. Таким образом, перебор можно рассматривать как результат работы нескольких вложенных циклов (их количество равно рангу массива *Array*), причем цикл для самого внутреннего измерения массива является внутренним, а для самого внешнего — внешним. Например, когда **a_loop** применяется к двухмерной матрице, проход осуществляется сперва по ее столбцам (во внутреннем цикле), а потом по строкам (во внешнем цикле). Поскольку *Loop* выполняется для каждого элемента массива, общее число итераций всегда равно **a_total** (*Array*). Результатом выполнения для **a_loop** является результат *Loop* на последней итерации (или (), если в массиве *Array* вообще нет элементов).

Таким образом, **a_loop** работает так, будто весь массив развернут в линейный вектор (фактически игнорируя при этом внутреннюю структуру массива). Если массив является операндом в стандартной операции вывода **f_put** — все его элементы выводятся в том же порядке, что и при применении к нему **a_loop** (как и при выводе списков, они выводятся подряд, без каких-либо дополнительных разделителей).

Загрузка и выгрузка массива

Имеется также возможность *выгрузить* все данные из массива в список (или наоборот — *загрузить* данные из списка в массив).

Загрузка и выгрузка элементов	
Нормальная форма	***Семантика***
a_save(Array)	*Возвращает список всех элементов в Array.*
a_load(Array, List)	*Загружает элементы из списка List в Array.*

Обе операции рассматривают массив как линейную последовательность элементов (упорядоченных так же, как и для **a_loop**). Функтор **a_save** (*Array*) возвращает все элементы массива *Array* в виде списка (сам массив при этом не меняется). Функтор **a_load** (*Array*, *List*) выполняет противоположную задачу: последовательно загружает элементы из списка *List* в массив *Array*.

Длина списка в обоих случаях (обычно) равна **a_total** (*Array*). Однако, она может быть и меньше, т. к. **a_save** отбрасывает все финальные пустые элементы. Если массив полностью пуст, **a_save** просто вернет (). Если аргумент *List* в **a_load** содержит больше элементов, чем

требуется — все избыточные элементы игнорируются.
Эти операции предназначены (в том числе) и для сохранения/восстановления хранящихся в массиве данных. Впрочем, как очевидно, метрику и ранг массива эти операции, конечно, не сохраняют (и не восстанавливают). При необходимости это можно сделать отдельно. Зная ранг, метрику и все элементы массива — нетрудно восстановить его в оригинальном виде.

Функциональная инициализация массива

Несколько менее тривиальный (по сравнению с **a_fill**) способ заполнения массива значениями элементов — использовать **a_init_all**:

Функциональная инициализация	
Нормальная форма	***Семантика***
a_init_all(Array, Func)	*Присвоить значения всем элементам массива Array, вычисляя их с помощью Func.*

Для каждого из элементов массива *Array* — вызывается функтор по ссылке *Func*, которому в качестве аргумента передается индекс (или список индексов) этого элемента (в том же порядке, как и для **a_elem**), и результат вычисления *Func* становится значением этого элемента. Например:

```
mtx1 = array (4, 4);
a_init_all (mtx1, !(x y) = (x*10 + y));
a_save (mtx1);
```

```
a_save:mtx1 => (0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23,
30, 31, 32, 33);
```

Возвращаемое значение для **a_init_all** всегда является пустым.

Копирование массива

Копирование массива — достаточно тривиальная операция:

Копия массива	
Нормальная форма	***Семантика***
a_copy(Array)	*Создает и возвращает копию массива Array.*

Результатом вызова **a_copy** будет массив с тем же рангом, метрикой, и с теми же элементами, что и массив *Array*. Собственно, на деле выполнение

```
ArrayB = a_copy (ArrayA);
```

— имеет тот же эффект, что и:

```
ArrayB = array (a_dims (ArrayA));
a_load (ArrayB, a_save (ArrayA));
```

— с той разницей, что **a_copy** выполняется намного быстрее, не используя промежуточного списка элементов, а копируя их напрямую.

Также заметим, что **a_copy** копирует массив, но не его элементы по отдельности! Т.е. если элементы массива являются мутабельными — то *ArrayA* и *ArrayB* будут разделять эти элементы между собой.

Массивы и списки

С точки зрения их функциональности, массивы и списки во многом взаимозаменямы. Вместо линейного массива может быть использован список (а вместо многомерного массива — набор из вложенных списков). У каждого из подходов есть свои сильные и слабые стороны.

Преимущества массива состоят в том, что они обеспечивают *более быстрый доступ* к своим элементам. Для списков время доступа к элементу *I* является линейной функцией от *I*, а для массива оно не зависит от индекса элемента. Другое преимущество массива заключается в том, что большой массив требует меньше памяти по сравнению со списком такого же размера.

Недостаток массива заключается в том, что вставлять в него (или удалять из него) элементы труднее. Все эти действия требуют физического перемещения данных массива — поэтому, когда массив достаточно большой, они могут потребовать определенного времени. Преимущество списка в том, что добавление новых элементов в любое его место (как и их удаление) выполняются быстро — и с одинаковой скоростью независимо от того, сколько элементов уже имеется в списке.

Достоинства массивов и списков можно комбинировать. Например, при чтении данных из некоторого источника (например, файла) можно сначала собрать их в список, а затем, когда количество элементов уже точно известно — скопировать в массив, для более быстрой работы с ними.

Заметим, что с чисто технической точки зрения массивы и списки — совершенно разные вещи. Так, никакие операции, определенные для массивов, к спискам не применимы. Напротив, с точки зрения всех списковых операций, массивы — *атомарные* элементы данных, не имеющие внутренней структуры. (Например, вызов *#Array* для любого массива *Array* всегда возвращает 1).

Словари

Создание словаря

Помимо массивов, язык дает возможность работать еще с одним контейнером: словарями. *Словарь* (или *хэш*) представляет собой неупорядоченный набор произвольных *значений*, связанных с *ключами*. При этом к каждому из значений, хранящихся в словаре, можно получить доступ по его ключу. Т.е. словарь можно рассматривать как массив неограниченного размера, вместо целых чисел индексированный произвольными элементами данных. (Каждую пару "ключ + значение" далее мы будем называть *элементом словаря*.) В процессе работы со словарем в него можно добавлять новые элементы, удалять существующие, и, разумеется, менять значения, связанные с ключами.
Имена функторов, связанных со словарями, по большей части начинаются с **h_**. Для создания нового словаря используется примитивный конструктор **hash** (или **h_create**).

Конструктор словаря	
Нормальная форма	***Семантика***
hash(Capacity, ...)	*Создает новый словарь (хэш) со внутренней емкостью Capacity*

Его результатом является новый словарь, а его параметрами (необязательными) является внутренняя емкость словаря (кол-во "карманов"), и (необязательно) минимум/максимум допустимой загрузки из расчета на один "карман". Эти параметры (обычно они опускаются) позволяют подстраивать эффективность хэша для конкретной задачи . Например:

```
Hash1 = hash (10);
Hash1;
```

```
set:(Hash1, hash:10) => <0 / 10>;
Hash1 => <0 / 10>;
```

Как легко видеть, словари обычно выводятся системой в виде <N / V>, где *N* — общее количество элементов в словаре, а *V* — его внутренняя емкость (т. е. кол-во «карманов» в нем).
Если при вызове **hash** параметры опущены — для них выбираются значения по умолчанию. Созданный обращением к **hash** словарь изначально всегда является пустым.

Проверка на словарь

Как и все проверки типов, она достаточно тривиальна:

Проверка на словарь	
Нормальная форма	***Семантика***
is_hash(Value)	*Истинно, если Value является словарем*

Выражение **is_hash**(V) возвращает истинное значение, если результат *V* — произвольный словарь (и ложное в противном случае).

Идентичность и неидентичность

При поиске ключа в словаре используется *сравнение на идентичность*. Операции, сравнивающие свои операнды на идентичность (или на неидентичность) доступны непосредственно (и вне связи со словарями):

Идентичность и неидентичность		
Синтаксис	***Нормальная форма***	***Семантика***
`[V == W]`	**`ident`**`(V, W)`	*Значение V идентично значению W?*
`[V <> W]` `[V ~= W]`	**`differ`**`(V, W)`	*Значение V НЕ идентично значению W?*

Операции *идентичности* и *неидентичности* — это самые «строгие» операции сравнения в языке. Так, если все скалярные сравнения допускают неявные приведения типов (предикаты **eq** и **ne** всегда приводят свои операнды к общему числовому типу, а предикаты **s_eq** и **s_ne** к строковому) — то операция **ident** не допускает никаких приведений! С точки зрения **ident**, скалярные значения 3, 3.0, “3” и “3.0” — являются *разными* (т.е. если операнды **ident** могут принадлежать к разным скалярным типам — их необходимо явно привести к какому-то одному). Заметим, что при сравнении строк также учитывается их тип: строки разного типа по определению считаются различными.

Однако, преимущество **ident** перед прочими операциями сравнения состоит в том, что она применима *к любым видам* данных. В частности, она применима к спискам: списки считаются идентичными, если их длины равны, и их элементы попарно идентичны. (Открытость/закрытость списка также принимается во внимание: открытый список *не идентичен* закрытому, даже если их элементы попарно совпадают!) Когда элементами списка являются другие списки, сравнение на идентичность применяется к ним рекурсивно.

Результат выполнения **differ** всегда противоположен таковому для **ident**. Заметим, что эти операции позволяют сравнивать произвольные данные на равенство — но не на упорядоченность. (В отличие от скаляров — для прочих разновидностей данных понятие об «упорядоченности» в языке не определено.)

Доступ к элементу словаря

Чаще всего работа со словарями выполняется *поэлементно*.

Доступ к элементу словаря		
Синтаксис	***Нормальная форма***	***Семантика***
`Hash->Key`	**`h_elem`**`(Hash, Key)`	*Доступ к элементу словаря Hash с ключом Key*

Эта операция также является аксессором: т. е. вычисление **h_elem** (*Hash*, *Key*) предоставляет мутабельный доступ к значению словаря *Hash*, связанному с ключом *Key*. Если элемента с таким ключом в словаре не было — он автоматически создается с (изначально) пустым значением. В качестве ключей в словаре могут использоваться выражения произвольного типа (в т. ч. любые скаляры и списки из скаляров). При поиске ключей применяется сравнение *на идентичность*.

Например:

```
Hash1->1 = 'one';
Hash1->2 = 'two';
Hash1->3 = 'three';
Hash1->'one' = 1;
Hash1->'two' = 2;
Hash1->'three' = 3;
```

позволяет занести в словарь *Hash1* ряд элементов, с заданными ключами и значениями. Поскольку в словаре может быть только один ключ с заданным значением, если в словаре уже присутствовали элементы с заданными ключами, их значения заменяются на новые. Значение элемента словаря (как и ключ) может иметь произвольный тип.
Однако, при использовании **h_elem** имеется проблема: этот функтор является не только аксессором, но (в определенных случаях) также и мутатором. А именно: если в словаре ранее не было элемента с запрошенным ключом — он автоматически *создается* (т.е. словарь при этом *изменяется*).И, даже если значение добавленного элемента не будет изменено — он все равно останется в словаре. Это не всегда безопасно: при обращении к большому количеству элементов с несуществующими ключами — словарь может очень сильно разрастись. Вот почему, когда необходим доступ к элементу только на чтение, лучше использовать примитив **h_lookup**.

Доступ (только чтение) к элементу словаря	
Нормальная форма	***Семантика***
h_lookup(Hash, Key)	*Чтение элемента словаря Hash с ключом Key*

Для **h_lookup** (в отличие от **h_elem**) если ключ *Key* отсутствует, он никогда не создается, а результатом в этом случае будет (). Например:

```
(h_lookup (Hash1, 1), h_lookup (Hash1, 2), h_lookup (Hash1,
3));
(h_lookup (Hash1, 'three'), h_lookup (Hash1, 'two'), h_lookup
(Hash1, 'one'));
```

```
(h_lookup:(Hash1, 1), h_lookup:(Hash1, 2), h_lookup:(Hash1,
3)) => ("one", "two", "three");
(h_lookup:(Hash1, "three"), h_lookup:(Hash1, "two"),
h_lookup:(Hash1, "one")) => (3, 2, 1);
```

Удаление элемента из словаря

После создания нового элемента словаря — он остается в нем независимо от того, какие значения ему присваиваются. В частности, значению элемента можно присвоить () — но и это автоматически не приводит к удалению элемента из словаря. Для этой цели следует использовать отдельный примитив: **h_remove**.

Удаление элемента словаря	
Нормальная форма	***Семантика***
h_remove(Hash, Key)	*Удаление элемента словаря Hash с ключом Key*

Выполнение **h_remove** (*Hash*, *Key*) удаляет из словаря *Hash* элемент с ключом *Key*

(возвращая результатом то значение, которое было с ним связано). Если элемента с таким ключом в словаре не было — ничего не изменяется, и результатом является ().

```
h_remove (Hash1, 1);
h_remove (Hash1, 2);
h_remove (Hash1, 3);
```

```
h_remove:(Hash1, 1) => "one";
h_remove:(Hash1, 2) => "two";
h_remove:(Hash1, 3) => "three";
```

Если необходимо быстро (и полностью) очистить словарь — удобнее всего использовать **h_clear**:

Очистка словаря	
Нормальная форма	***Семантика***
h_clear(Hash)	*Полная очистка Hash, с удалением всех элементов*

Конечно, вызов **h_clear** выполняется намного быстрее, чем уничтожение элементов по одному с помощью **h_remove**. Наверное излишне напоминать, что это — крайне деструктивная операция, которую нужно использовать лишь тогда, когда вы уверены в том, что делаете.

В качестве результата, **h_clear** возвращает *число* элементов, присутствовавших в словаре *до очистки*.

Проверка присутствия ключа

Для этой цели имеется встроенный предикат **h_exists**:

Проверка присутствия элемента словаря	
Нормальная форма	***Семантика***
h_exists(Hash, Key)	*Проверяет присутствие в Hash элемента с ключом Key*

Здесь все весьма тривиально: результатом является 1 (*истина*), если в словаре *Hash* присутствует элемент с ключом *Key* (с произвольным, в т. ч. и пустым, значением) — или 0 (*ложь*) в противном случае.

Количество элементов

Количество элементов в словаре	
Нормальная форма	***Семантика***
h_count(Hash)	*Всего элементов в Hash*

Этот функтор возвращает текущее количество элементов, хранящихся в словаре *Hash*.

Итераторы по словарю

Для всех контейнеров данных определены свои итераторы, и словари — не исключение. Перебор всех элементов словаря выполняет **h_loop**.

Итератор по словарю	
Нормальная форма	***Семантика***
h_loop(Hash, Var, Body)	*Выполнить Body для всех элементов Hash.*

Итератор **h_loop** выполняет (или вычисляет) свой операнд *Body* для *каждого* элемента словаря *Hash*, перед этим присвоив операнду *Var* (он должен быть мутабельным) ключ и значение данного элемента, в виде списка (*Ключ*, *Значение*). Хотя итератор гарантированно выполняется для каждого элемента словаря (т. е. ровно **h_count** (*Hash*) раз), но точный порядок их перебора не определен. В качестве значения **h_loop** (как и большинство других итераторов) возвращает последнее значение *Body* (или (), если словарь *Hash* был пустым).
Например, таким вот образом можно вывести все элементы словаря (для *Hash1* из предыдущего раздела):

```
h_loop (Hash1, elem, <: (l_head(elem), " => ", l_tail(elem),
"\n"));
```

```
two => 2
three => 3
one => 1
```

Если словарь является операндом в стандартной операции вывода **f_put** — все его элементы выводятся в том же порядке, что и при применении к нему **h_loop**. (Они выводятся подряд, без каких-либо дополнительных разделителей).

Загрузка и выгрузка элементов

Имеются (как и для массивов) примитивы, позволяющие преобразовывать словари в списки ключей и/или значений (и наоборот).

Выгрузка из словаря	
Нормальная форма	***Семантика***
h_save(Hash)	*Возвращает содержимое Hash в виде списка.*
h_keys(Hash)	*Возвращает все ключи из Hash в виде списка.*
h_values(Hash)	*Возвращает все значения из Hash в виде списка.*
h_load(Hash, Elems)	*Загружает в Hash значения Elems.*

Функтор **h_save** (*Hash*) возвращает содержимое словаря *Hash* в виде списка (всегда открытого), элементами которого (как для **h_loop**) являются списки вида (*Ключ*, *Значение*). Если нужен получить только список *ключей* словаря, проще использовать примитив **h_keys** (*Hash*); если нужен только список *значений* — примитив **h_values** (*Hash*). (Оба этих функтора также всегда возвращают открытые списки). Порядок элементов во всех случаях следует считать *неопределенным* (однако, он всегда одинаков для **h_keys**, **h_values** и **h_save**,).
Например, для рассмотренного выше *Hash1*:

```
h_keys(Hash1);
h_values(Hash1);
h_save(Hash1);
```

```
h_keys:Hash1 => ("one", "three", "two", );
h_values:Hash1 => (1, 3, 2, );
h_save:Hash1 => (("one", 1), ("three", 3), ("two", 2), );
```

Функтор **h_load** выполняет операцию, обратную **h_save**. А именно, получая в качестве своего аргумента *List* открытый список пар (*Ключ*, *Значение*) — **h_load**(*Hash*, *List*) загружает их все, как элементы, в словарь *Hash.* Например:

```
h_load (Hash1,
   ['four' 4],
   ['five' 5],
   ['six' 6],
   );
```

```
h_load:(Hash1, ("four", 4), ("five", 5), ("six", 6), ) =>
<6 :: 10>;
```

```
h_save (Hash1);
```

```
h_save:Hash1 => (("six", 6), ("one", 1), ("three", 3),
("four", 4), ("two", 2), ("five", 5), );
```

Как легко видеть, старое содержимое словаря остается нетронутым. Однако, если в списке *List* присутствует уже имеющийся в словаре ключ — новое значение для этого ключа затирает старое. В остальном, порядок элементов в **h_load** не имеет значения. Обычно элементами списка *List* являются списки, но если в этот список включен атомарный элемент *Atom*, то он рассматривается как (*Atom*,) — т. е. в словарь будет добавлен элемент с ключом *Atom* и пустым значением.

Реструктурирование словаря

В завершение, мы рассмотрим встроенный функтор **h_rehash** — который изменяет внутреннюю организацию словаря, но не меняет его содержимое.

Реконструктор словаря	
Нормальная форма	***Семантика***
h_rehash(Hash, Capacity, ...)	*Реконструирует словарь Hash, изменяя его внутреннюю емкость на Capacity*

Внутренняя емкость словаря *Hash* (от которой зависит в т.ч. и эффективность поиска элементов в нем) с помощью этой операции может быть изменена *принудительно*. Параметр *Capacity* имеет тот же смысл, что и для конструктора **h_create** — но, в отличие от него, здесь не создается новый словарь, а изменяется уже существующий. При этом, все данные, имевшиеся в словаре *Hash*, остаются в целости и сохранности. Правда, их внутренний порядок (т. е. тот, в котором они перечисляются в **h_loop** или **h_save**) — скорее всего, изменится.

Еще один эффект (в основном, побочный) от **h_rehash** — *восстановление внутренней*

целостности словаря, если она вдруг была нарушена. Такое происходит редко — но все-таки может произойти.
Дело в том, что многие операции (такие, как **h_loop**, **h_save** или **h_keys**) предоставляют *прямой доступ* к ключам словаря. Когда ключи являются скалярами, это не вызывает каких-либо проблем — но ключами могут быть и списки, и другие мутабельные структуры данных. *Мутабельность ключей* означает, что при желании их можно изменить. Бесцеремонное изменение мутабельного ключа в словаре — обычно приводит к тому, что соответствующий элемент перестает находиться операциями-аксессорами, такими как **h_elem** или **h_lookup**. Т.е. внутренняя целостность словаря при этом нарушается. Однако, любое применение **h_rehash** к «испорченному» словарю успешно восстанавливает его целостность, вновь делая все его элементы доступными по ключам. Итак, хотя прямое изменение мутабельных ключей не рекомендуется — в крайних случаях делать это можно, надо лишь не забыть потом реструктурировать словарь.
В качестве результата — **h_rehash** возвращает текущее число элементов в словаре.

Вычисление хэш-кода

Со словарями имеет непосредственную связь и эта операция:

Внутренний хэш-код значения	
Нормальная форма	***Семантика***
hcode(Value)	*Внутренний числовой хэш-код для значения Value*

Значение *Value* может быть совершенно любым. Результатом выполнения **hcode** является некое целое число, представляющее *хэш-код* своего операнда. Внутри интерпретатора эти хэш-коды используются для хэширования ключей в словарях — однако, их можно использовать и для многих других целей.
Реализация этого функтора всегда гарантирует, что для *идентичных* значений хэш-коды также будут равны: т. е. из *value1* [==] *value2* всегда следует, что **hcode**(*value1*) == **hcode**(*value2*). Помимо этого, никаких особых гарантий относительно хэш-кода языком не дается. В частности, не гарантируется, что алгоритм **hcode** останется неизменным между реализациями языка — поэтому для постоянно хранимых где-либо (*persistent*) данных этот алгоритм не годится. Тем более, **hcode** не предназначен быть адекватной заменой для «серьезных», т. е. криптографически устойчивых алгоритмов вычисления хэш-кодов — таких, как MD5 или SHA-256.

Образцы

В языке предусмотрен механизм *образцов*, или *регулярных выражений*. (Оба этих понятия совершенно синонимичны — но далее мы будем использовать термин «образцы», как более короткий.) Образцы предлагают удобный способ для того, чтобы (вместо жестко фиксированной строки) задать некое семейство строк, подчиняющихся определенному *шаблону*.

Механизм регулярных выражений (встроенный или библиотечный) предусмотрен в большинстве языков. Но, как правило, в них регулярные выражения задаются в виде символьных строк определенного формата (т. е. фактически, для регулярных выражений вводится *свой собственный* язык, никак не связанный с основным языком). В AWL образцы являются самостоятельным и полноправным типом данных, который является важной частью языка. Как и большинство нетривиальных структур данных, образцы создаются с помощью встроенных функторов-*конструкторов*. Для многих из них операндами являются другие образцы. Самих конструкторов образцов в языке немного, и они довольно примитивны по сути — но, комбинируя их, можно строить довольно сложные шаблоны. Заметим также, что образец, созданный один раз, может использоваться многократно: как и все данные, он будет существовать, пока имеется хотя бы одна ссылка на него.

Основные операции, определенные для образцов — различные виды *сопоставления* и *поиска*. Они также в основном реализованы в виде встроенных функторов. (Их имена обычно начинаются с префикса **rx_**.)

Литеральные образцы

Простейшие операции-конструкторы позволяют создавать образцы, соответствующие как отдельным фиксированным символам, так и символьным строкам.

Конструкторы образцов-литералов	
Нормальная форма	***Семантика***
rx_char(Code)	*Сопоставляется с литеральным символом с кодом Code*
rx_string(String)	*Сопоставляется с литеральной строкой String*

Конструктор **rx_char**(*Code*) возвращает примитивный образец, соответствующий одному-единственному символу с кодом *Code*. Аналогично, конструктор **rx_string**(*String*) возвращает примитивный образец, соответствующий только строке *String*.

Например:

```
rx_char (48);
rx_string ("Hello");
```

```
rx_char:48 => '0';
rx_string:"Hello" => "Hello";
```

На самом деле, обе эти операции довольно редко нужны в явном виде. Дело в том, что большинство операций, работающих с образцами — умеют применять их *неявно*. Если в качестве операнда требуется образец, а вместо него передается строка — к ней автоматически применяется конструктор **rx_string**. Если же вместо образца передается число — к нему автоматически применяется конструктор **rx_char**. Таким образом, строковые

и числовые значения автоматически приводятся к образцам (прочие типы данных как замена образцов недопустимы). Заметим, что функторы для образцов интерпретируют числовые значения не так, как скалярные. Если для строковых (скалярных) операций число 34 преобразуется в строку «34», то для образцов оно будет преобразовано в строку «"» (т. к. в ASCII и Unicode код двойной кавычки равен 34). Если же при работе с образцом явно нужна именно «скалярная» семантика — можно использовать вызов **string**.

Произвольный символ

Тривиальный образец, конструктор которого не имеет аргументов.

Произвольный символ	
Нормальная форма	***Семантика***
rx_any()	*Сопоставляется с любым символом*

Этот образец сопоставляется с (одним) произвольным символом, без каких-либо ограничений.

Классы символов

Менее тривиальный вид образца — определяет не конкретный символ, а целый их *класс*.

Классы символов	
Нормальная форма	***Семантика***
rx_any_in(Pred)	*Сопоставляется с каждым символом, для которого предикат Pred истинен.*
rx_any_ex(Pred)	*Сопоставляется с каждым символом, для которого предикат Pred ложен.*

Как видим, два конструктора **rx_any** различаются только полярностью условия. Сами классы символов задаются с помощью *предикатов*, которые мы уже рассматривали выше. Напомним, что их операндом всегда является *код* проверяемого символа (т. е. число), а результатом — 1 (если символ относится к заданному классу) или 0 (в противном случае). Разумеется, можно использовать любые встроенные классы:

```
rx_any_in (!cc_digit)
```

— сопоставляется с любой десятичной цифрой;

```
rx_any_ex (!cc_alpha)
```

— сопоставляется со всеми символами, *кроме* алфавитно-цифровых.
В качестве операнда также можно использовать и конструкторы для классов символов (**cc_incl** и **cc_excl**), где класс определяется строкой-операндом:

```
rx_any_in (cc_incl ("aeiou"))
```

— сопоставляется со строчными буквами *«a»*, *«e»*, *«i»*, *«o»*, *«u»* (и только с ними).
Заметим, что поскольку и **rx_any_** и **cc_** позволяют выбирать полярность, здесь есть определенная избыточность. На самом деле, **rx_any_in** (**cc_incl**(*String*)) всегда равносильно

rx_any_ex (**cc_excl**(*String*)). Аналогично, и **rx_any_ex** (**cc_incl**(*String*)) всегда равносильно **rx_any_in** (**cc_excl**(*String*)).

Сопоставление с образцом

Основное предназначение образцов, конечно, состоит в том, что их можно *сопоставлять* с символьными строками. (Далее, строку, с которой происходит сопоставление — мы будем называть *субъектной*.) Простейший вид сопоставления — это проверка соответствия заданному образцу начала строки, для чего используется примитив **rx_match**.

Сопоставление с образцом	
Нормальная форма	***Семантика***
rx_match(Pattern, String)	*Сопоставляет строку String с образцом Pattern*

Результатом для **rx_match**(*Pattern*, *String*) является *длина* начального фрагмента строки *String*, который удалось успешно сопоставить с *Pattern* (т. е. число символов). Если сопоставить образец не удалось, то результатом является -1. Заметим, что эту ситуацию необходимо отличать от *успешного* сопоставления *пустой* строки — при котором результатом, конечно, будет 0.
Примитив **rx_match** проверяет на соответствие *Pattern* только начало строки *String* (если образец в строке присутствует, но не в самом начале — сопоставление будет неудачным).

Поиск образца в строке

Чаще употребляются поисковые операции: **rx_findfirst** и **rx_findlast**.

Поиск образца	
Нормальная форма	***Семантика***
rx_findfirst(Pattern,String)	*Найти первый образец Pattern в строке String*
rx_findlast(Pattern,String)	*Найти последний образец Pattern в строке String*

Они выполняют поиск образца *Pattern* в заданной строке *String* (для **rx_findfirst** поиск осуществляется *вперед* с начала; для **rx_findlast** — *назад* с конца). Для обеих операций их результатом (*Range*) является *диапазон индексов*, определяющих местонахождение обнаруженного контекста в строке *String* (т. е. отрезок строки *String* $[*Range*] полностью сопоставляется с образцом *Pattern*). Если образец в строке не найден — в качестве результата возвращается пустое значение ().
Заметим, что эти операции во многом похожи на свои чисто строковые аналоги: **s_findfirst** и **s_findlast**. Основное различие заключается в том, что строковые операции поиска возвращают *числовой индекс* (смещение найденной подстроки в строке). Для строковых операций поиска возвращать *длину* найденного контекста не нужно: она всегда равна длине «разыскиваемой» подстроки. Но для образцов эта длина может быть не фиксированной — поэтому операции поиска образца возвращают не просто индекс, а диапазон индексов.

Альтернация образцов

Все рассмотренные выше конструкторы образцов были *примитивными*, и создавали новый образец из простых операндов: кодов символов, строк, предикатов. Рассмотрим теперь менее тривиальные конструкторы, создающие сложные образцы из более простых.
Операция *альтернации* позволяет создать образец, сопоставляющийся с одной из двух более

простых альтернатив.

Альтернация образцов	
Нормальная форма	***Семантика***
rx_alt(Pat1, Pat2)	*Сопоставляется с Pat1 или с Pat2*

Результат **rx_alt**(*Pat1*, *Pat2*) — образец, сопоставляющийся либо с *Pat1*, либо с *Pat2*. Например:

```
rx_alt (“Hello”, “Goodbye”);
```

создает образец, сопоставляющийся со строкой “Hello” или “Goodbye” (и ни с чем другим).

```
rx_alt:("Hello", "Goodbye") => <"Hello" | "Goodbye">;
```

Разумеется, вложенные альтернации позволяют создать образец, сопоставляющийся с произвольным количеством альтернатив. Например:

```
rx_alt (“Yes”, rx_alt (“No”, “Maybe”));
```

создает образец, сопоставляющийся со строками “Yes”, “No” и “Maybe”.

```
rx_alt:("Yes", rx_alt:("No", "Maybe")) => <"Yes" | <"No" |
"Maybe">>;
```

Использовать вложенные альтернации иногда неудобно (особенно когда их много). Для краткости, можно определить конструктор образца для сопоставления со списком альтернатив:

```
! rx_alt_list (list) =
(#list > 1) ? rx_alt (l_head(list), rx_alt_list(l_tail(list)))
: list;
```

Теперь предыдущий образец можно компактнее определить так:

```
rx_alt_list (“Yes”, “No”, “Maybe”);
```

Приведем пример простого поиска для многовариантного образца:

```
rx_findfirst (rx_alt ('ab', 'ba'), 'abbbbba');
rx_findlast (rx_alt ('ab', 'ba'), 'abbbbba');
```

```
rx_findfirst:(rx_alt:("ab", "ba"), "abbbbba") => (0, 2);
rx_findlast:(rx_alt:("ab", "ba"), "abbbbba") => (5, 7);
```

Конкатенация образцов

Операция *конкатенации* позволяет создать новый образец, успешно сопоставляющийся с последовательностью идущих подряд двух образцов-операндов.

Конкатенация образцов	
Нормальная форма	***Семантика***
rx_cat(PatL, PatR)	*Сопоставляется с последовательностью PatL, PatR*

Результат **rx_cat**(*Pat1*, *Pat2*) — образец, сопоставляющийся с любой конкатенацией образцов *Pat1* и *Pat2*. Так, например,

```
rx_cat (rx_alt (“hello”, “goodbye”), “!”);
```

возвращает образец, сопоставляющийся с строкой “hello!” или “goodbye!”. Аналогично, результатом для:

```
rx_cat (rx_alt (“aa”, “bb”), rx_alt (“cc”, “dd”));
```

является образец, сопоставляющийся с одной из четырех строк: “aacc”, “bbcc”, “aadd”, “bbdd”.

```
rx_cat:(rx_alt:("aa", "bb"), rx_alt:("cc", "dd")) => <<"aa"
| "bb"> & <"cc" | "dd">>;
```

Вкладывая конкатенации друг в друга — можно определить образец, сопоставляющийся со сколь угодно длинной последовательностью, составленной из более простых. Как и для альтернации, можно определить простую операцию для сцепления списка образцов:

```
! rx_cat_list (list) =
(#list > 1) ? rx_cat (l_head(list), rx_cat_list(l_tail(list)))
: list;
```

Применение конкатенации образцов обычно не имеет смысла, когда оба операнда являются символьными или строковыми литералами. В этом случае, вместо **rx_cat** стоит использовать **s_cat**: она возвращает готовую строку, сопоставление с которой осуществляется быстрее. Однако, в некоторых случаях (например, когда конкатенируются строки разного типа) отказ от **s_cat** в пользу **rx_cat** может несколько сэкономить память (ценой эффективности).

Репликация образца

Операция *репликации* позволяет создать новый образец, сопоставляющийся с повторением образца-операнда (т.е. его конкатенацией с самим собой) некоторое (не обязательно фиксированное!) число раз.

Репликация образца	
Нормальная форма	***Семантика***
rx_rep(Range, MinMax, Pattern)	*Сопоставляется с повторением образца Pattern Range раз (MinMax определяет максимум или минимум повторов)*

Результатом для **rx_rep** (*Range*, *MinMax*, *Pattern*) является образец, сопоставляющийся с последовательностью повторений операнда-образца *Pattern*. Допустимое число повторений задается первым параметром, диапазоном *Range*: *минимальное* число повторений задается нижней границей диапазона, *максимальное* — верхней границей. Если нижняя граница

диапазона опущена — предполагается 0; если опущена верхняя граница — максимальное число повторений не ограничено (вплоть до бесконечности). Если операнд *Range* является пустым, количество повторений может быть любым (включая 0). Второй параметр *MinMax* рассматривается как логическое значение: при его ложности, приоритет отдается *минимальному* количество повторений, при его истинности — *максимальному*. Другими словами, если *MinMax* истинно, то репликация является *жадной* (ищется самая длинная последовательность в пределах диапазона), а если ложно — то она является *скромной* (ищется самая короткая последовательность). Однако, в обоих случаях число повторений не может быть меньше нижней границы диапазона, или больше его верхней границы.
Приведем примеры:

```
rx_rep (2..5, 0, "()");
  rx_rep:((2, 5), 0, "()") => <"()" * 2.>>.5>;
```

Образец-результат сопоставляется с повторением строки “()”: как минимум 2 раза, и как максимум 5 раз подряд. (При этом предпочтение отдается минимуму повторов.)

```
rx_rep (10, 1, "/-/");
  rx_rep:(10, 1, "/-/") => <"/-/" * 0.<<.10>;
```

Результат сопоставляется с повторением строки “/-/” не более 10 раз подряд (нижний лимит числа повторов отсутствует), предпочтение отдается максимуму повторов.
Этот образец часто употребляется для поиска последовательностей повторяющихся символов. Следующие виды репликации нужны так часто, что для них тоже есть смысл определить свои конструкторы:

```
! rx_rep_any (Range, MinMax) = rx_rep (Range, MinMax,
rx_any());
```

Сопоставляется с последовательностью любых символов, с длиной в диапазоне *Range*, и с «жадностью», заданной *MinMax*. Например, **rx_rep_any**(10..20, 0) возвращает образец, сопоставляющийся с произвольной (предпочтительно, короткой) последовательностью не менее 10 и не более 20 символов длиной.
Аналогично:

```
! rx_rep_in (Range, MinMax, Pred) = rx_rep (Range, MinMax,
rx_any_in(Pred));

! rx_rep_ex (Range, MinMax, Pred) = rx_rep (Range, MinMax,
rx_any_ex(Pred));
```

Это конструкторы для последовательности символов, определяемой предикатом *Pred*, позитивно (**rx_rep_in**) или негативно (**rx_rep_ex**).
Например, **rx_rep_in** (5..10, 1, !**cc_digit**) сопоставляется с (самой длинной) последовательностью из не менее 5, и не более 10 десятичных цифр. Аналогично — **rx_rep_ex** (16, 0, !**cc_blank**) сопоставляется с (самой короткой) последовательностью из не более 16 *не*пробельных символов.
Заметим, что сопоставление с репликацией примитивных образцов (определенный символ или строка, класс символов, произвольный символ) выполняется весьма эффективно.

Репликация более сложных образцов «работает» медленнее.

Оценка длины образца

Имеется полезная операция, позволяющая быстро оценить, со строкой какой длины может быть успешно сопоставлен заданный образец.

Длина сопоставления	
Нормальная форма	***Семантика***
rx_length(Pattern)	*Диапазон длин для строки, с которой можно сопоставить образец Pattern*

Результатом для **rx_length**(*Pattern*) является список-диапазон, границы которого определяют *минимальную* и *максимальную* длину контекста, с которым может быть успешно сопоставлен образец *Pattern*. Если образец может быть сопоставлен с пустой строкой, нижней границей диапазона является 0. Если же образец может быть сопоставлен со сколь угодно длинной строкой (вплоть до бесконечности), верхняя граница диапазона пуста.
Например:

```
rx_length (rx_alt ('aa', 'bbcc'));
rx_length (rx_cat (rx_alt ('hello', 'goodbye'), '.'));
rx_length (rx_rep (4..7, 0, rx_alt ('ad', 'ret')));
```

```
rx_length:rx_alt:("aa", "bbcc") => (2, 4);
rx_length:rx_cat:(rx_alt:("hello", "goodbye"), ".") => (6,
8);
rx_length:rx_rep:((4, 7), 0, rx_alt:("ad", "ret")) => (8,
21);
```

Для тривиальных образцов результат этой операции всегда легко предсказуем. Так, для образцов **rx_char**, **rx_any**, **rx_any_in**, **rx_any_ex —** всегда возвращается (1, 1). Для образца-фиксированной строки (*String*) — возвращается (#$*String*, #$*String*). Нижняя и верхняя границы различаются только для нетривиальных образцов.

Позиционные якоря

Рассмотрим теперь так называемые «якоря» (“anchors”). Это — общее название для всех образцов, которые сопоставляются с *пустой строкой*, находящейся в определенной (не всякой) позиции субъектной строки.

Позиционные якоря	
Нормальная форма	***Семантика***
rx_at_start(StartOff)	*Образец-якорь для позиции StartOff от начала строки*
rx_at_end(EndOff)	*Образец-якорь для позиции EndOff от конца строки*

Позиционные якоря позволяют «привязывать» образец к определенной позиции в строке, с которой происходит сопоставление. Образец **rx_at_start**(*StartOff*) сопоставляется с пустой строкой в позиции *StartOff* субъектной строки относительно ее начала (вперед). Образец **rx_at_end**(*EndOff*) сопоставляется с пустой строкой в позиции *EndOff* субъектной строки

относительно ее конца (назад). Заметим, что значения параметров *StartOff* и *EndOff* всегда должны быть неотрицательными (отрицательные не имеют смысла).
Как важные частные случаи, **rx_at_start**(0) является «якорем» для *начала* строки, а **rx_at_end**(0) — «якорем» для *конца* строки. Использование этих якорей позволяет привязать образец к началу или к концу.

Границы контекста

Эти якоря сопоставляются с *началом* или с *концом* последовательности символов, принадлежащих определенному классу (заданному предикатом).

Граничные якоря	
Нормальная форма	***Семантика***
rx_before_in(Pred)	*Начало последовательности, позитивной для Pred?*
rx_before_ex(Pred)	*Начало последовательности, негативной для Pred?*
rx_after_in(Pred)	*Конец последовательности, позитивной для Pred?*
rx_after_ex(Pred)	*Конец последовательности, негативной для Pred?*

Здесь *Pred* — ссылка на функтор-предикат, определяющий класс символов так же, как и для **rx_all_in** и **rx_all_ex** (допускаются любые встроенные или явно определенные предикаты). «Позитивным» контекстом мы будем называть последовательность символов, удовлетворяющих предикату *Pred*, а «негативным» — не удовлетворяющих ему.
Якоря **rx_before** сопоставляются с *началом контекста*: т. е. следующий символ должен быть включенным в контекст (**before_in**) / исключенным из него (**before_ex**), а предыдущий (соответственно) должен или отсутствовать (начало строки), или быть исключенным (для **_in**) / включенным (для **_ex**). Аналогично, якоря **rx_after** сопоставляются с *концом контекста*: предыдущий символ должен быть включен в контекст (**after_in**) / исключен из него (**after_ex**), а следующий (соответственно) должен или отсутствовать (конец строки), или же быть исключенным (для **_in**) / включенным (для **_ex**).
Вот пара примеров:

```
rx_before_in (!cc_digit);
```

— «якорь» для начала последовательности десятичных цифр;

```
rx_after_ex (!cc_blank);
```

— «якорь» для конца последовательности любых непробельных символов;

```
rx_after_in (cc_incl ("xyz"));
```

— «якорь» для конца последовательности из символов “x”, “y” или “z”.
Заметим, что (по определению) **rx_before** *никогда* не сопоставляется с концом строки (а **rx_after —** с ее началом). Также заметим, что если бы не наличие дополнительной проверки на начало/конец строки — то были бы полными синонимами **rx_before_in** и **rx_after_ex** (равно как и **rx_before_ex** и **rx_after_in**).

Предшествующий/следующий контекст

Эти образцы-якоря обеспечивают проверку на присутствие / отсутствие некой заданной строки *перед* текущей позицией в субъектной строке, или *после* нее.

Контекстные якоря	
Нормальная форма	***Семантика***
rx_is_before(Ctx)	*Контекст Ctx присутствует перед текущей позицией?*
rx_not_before(Ctx)	*Контекст Ctx отсутствует перед текущей позицией?*
rx_is_after(Ctx)	*Контекст Ctx присутствует после текущей позиции?*
rx_not_after(Ctx)	*Контекст Ctx отсутствует после текущей позиции?*

Осмысленное применение этих якорей предполагает, что строка Ctx не пуста (т. к. для пустой строки, сопоставление с **rx_is_(before|after)** заведомо удачно, а с **rx_not_(before|after)** заведомо неудачно. Заметим, что в любом случае сам контекст *Ctx никогда не становится* частью сопоставления.

Сочетание якорей

В нетривиальных случаях несколько образцов-якорей можно комбинировать с помощью определенных логических операций. Например, *альтернация якорей* позволяет реализовать над ними логическую операцию «ИЛИ»:

```
rx_alt (rx_at_start(8), rx_before_in(!cc_alpha));
```

Результат — образец, задающий или восьмую позицию относительно начала субъектной строки, или начало последовательности алфавитных символов. (При этом сам он — как и все якоря — сопоставляется с пустой строкой).
Примерно таким же образом — *конкатенация якорей* успешно реализует над ними логическую операцию «И».

```
rx_cat (rx_at_end(10), rx_is_before("abc"));
```

Этот образец сопоставляется с позицией за десять символов до конца строки, причем только в случае, когда этой позиции предшествует строка «abc». Если это не так, сопоставление неудачно.

Пустой образец и неудача

Имеются два очень тривиальных образца.

Пустой образец	
Нормальная форма	***Семантика***
rx_null()	*Сопоставляется с пустой строкой.*

Этот образец всегда успешно сопоставляется с пустой строкой, где бы она не находилась.

Неудачный образец	
Нормальная форма	***Семантика***

Неудачный образец	
rx_fail()	*Не сопоставляется ни с чем.*

Полная противоположность предыдущему: результатом будет образец, сопоставление с которым *всегда* завершается неудачно.
(Оба образца более полезны для целей отладки, чем для каких-либо реальных задач.)

Регистры и сопоставление с ними

В процессе сопоставления строки с образцом могут быть задействованы *регистры*. Регистры — это нумерованные ячейки временной памяти, содержащие определенные фрагменты субъектной строки. По умолчанию, доступно 4 регистра (которые нумеруются с нуля). Перед началом всякого сопоставления, регистры считаются пустыми (т. е. Не установленными).
Вот два конструктора образцов, использующих регистры:

Образцы, использующие регистры	
Нормальная форма	***Семантика***
rx_store(RegNo, Pattern)	*Аналогично образцу Pattern — но при успешном сопоставлении, результат записывается в RegNo*
rx_recall(RegNo)	*Сопоставляется с подстрокой, уже записанной в регистр RegNo*

Итак, образец **rx_store** представляет собой своеобразную настройку над своим операндом, произвольным образцом *Pattern*. Сопоставление с результатом происходит так же, как и с *Pattern* — но при удаче, сопоставленный фрагмент субъектной строки записывается *в регистр* с номером *RegNo*. Обращение к **rx_recall** создает образец с противоположной семантикой: он сопоставляется только с текущим содержимым регистра *RegNo*. Сопоставление неудачно, если в регистр ничего не было записано.
Использование **rx_store** и **rx_recall** позволяет легко создать образец, содержащий повторяющиеся (неизвестные заранее) элементы.

Множественный поиск

Мы рассмотрим теперь семейство функторов, работающих не с одним найденным в строке образцом, а с целым их набором. Простейшей из них является операция *множественного поиска* **rx_locate**: она (в отличие от **rx_findfirst** / **rx_findlast**) позволяет найти сразу *несколько* вхождений заданного образца в строку.

Множественный поиск образца в строке	
Нормальная форма	***Семантика***
rx_locate(Pattern, String, Range, Direction)	*Поиск образца Pattern в строке String (ищутся Range вхождений в направлении Direction)*

Результатом является список из найденных в строке *String* вхождений образца *Pattern* (он всегда открытый, т. е. имеет вид *Found1*, *Found2*, … *FoundN*,). Каждый из элементов этого списка является *диапазоном индексов*, определяющим, где именно в строке найден образец (так же, как для **rx_findfirst** и **rx_findlast**). Рассмотрим параметры функтора подробнее.

Смысл первых двух параметров очевиден: они задают образец *Pattern* для поиска в строке, и саму строку *String*, в которой выполняется поиск. Последний параметр *Direction* задает направление поиска: *вперед с начала* строки (если **false**) или *назад с конца* строки (если **true**). Наконец, параметр *Range* в общем случае является *диапазоном* значений. Его *верхняя граница* задает максимальное число вхождений в строке (т. е. больше, чем задано, никогда не ищется), а его *нижняя граница* задает количество первых найденных вхождений, которые *отбрасываются* (т. е. не будут включены в список-результат). (Заметим, что при поиске назад — *первые* найденные значения будут физически *последними* в строке *String*.) Любая из границ *Range* может отсутствовать: если отсутствует нижняя граница, то никакие начальные вхождения не отбрасываются; если отсутствует верхняя — лимита на количество вхождений нет вообще. Таким образом, в списке никогда не может быть больше, чем *Range*[1] - *Range*[0] элементов (но, конечно, может быть меньше, т. к. в субъектной строке может найтись меньше образцов, чем требуется).

Вот несколько примеров вызова **rx_locate**:

- **rx_locate** (*Pattern*, *String*, 0..1, 0): Ищется только самое первое вхождение
- **rx_locate** (*Pattern*, *String*, 0..1, 1): Ищется только самое последнее вхождение
- **rx_locate** (*Pattern*, *String*, 0..*n*, 0): Ищется *n* первых вхождений
- **rx_locate** (*Pattern*, *String*, 0..*n*, 1): Ищется *n* последних вхождений
- **rx_locate** (*Pattern*, *String*, *n*.., 0): Ищутся все вхождения, кроме *n* первых
- **rx_locate** (*Pattern*, *String*, *n*.., 1): Ищется все вхождения, кроме *n* последних
- **rx_locate** (*Pattern*, *String*, (), 0): Ищутся вообще все вхождения
- **rx_locate** (*Pattern*, *String*, (), 1): (то же самое)
- **rx_locate** (*Pattern*, *String*): (то же самое)

В простейшем случае, когда оба последних параметра опущены — ищутся все вхождения образца в строке, сколько б их не было.

```
rx_locate ('ain', "The rain in Spain falls mainly on a
plain");
```

```
rx_locate:("ain", "The rain in Spain falls mainly on a
plain") => ((5, 8), (14, 17), (25, 28), (38, 41), );
```

Разрезание строки

Функтор **rx_split** позволяет разрезать субъектную строку в тех местах (всех, или некоторых из них), где найден образец. (Под «разрезанием» мы имеем в виду то, что возвращается результат, состоящий из отрезков субъектной строки — сама строка, конечно, не меняется.)

Разрезание строки по образцу	
Нормальная форма	***Семантика***
rx_split(Pattern, String, Range, Direction)	*Разрезает строку String там, где найден Pattern (ищутся Range вхождений в направлении Direction)*

Все параметры интерпретируются совершенно так же, как и для **rx_locate**. Отличается результат: он является списком отрезков строки, заключенных *между* найденными вхождениями *Pattern* (сами вхождения исключаются). Параметры *Range* и *Direction* позволяют «фильтровать» список вхождений так же, как и при вызове **rx_locate**. Если они опущены — из субъектной строки «вырезаются» все найденные вхождения.

Например:

```
rx_split ('ain', "The rain in Spain falls mainly on a plain");
```

```
rx_split:("ain", "The rain in Spain falls mainly on a
plain") => ("The r", " in Sp", " falls m", "ly on a pl",
"");
```

Заметим, элементов в списке-результате всегда на единицу больше, чем найденных образцов. Начальные и/или конечные пустые строки тоже включаются в результат.

Замена образцов в строке

Функтор **rx_replace** позволяет заменить в субъектной строке все (или избранные) вхождения образца на что-либо иное (здесь опять, под «заменой» мы имеем в виду возвращаемый результат — сама строка не изменяется).

Замена образцов	
Нормальная форма	***Семантика***
rx_replace(Pattern, String, Replacer, Range, Direction)	*Заменить вхождения Pattern в String на Replacer (ищутся Range вхождений в направлении Direction)*

Здесь параметры *Range* и *Direction* также имеют такой же смысл, что и в вызове **rx_locate**. Если они опущены, заменяются все найденные вхождения образца в строке. Например:

```
rx_replace ('ain', '<=>', "The rain in Spain falls mainly on a
plain");
```

```
rx_replace:("ain", "<=>", "The rain in Spain falls mainly
on a plain") => "The r<=> in Sp<=> falls m<=>ly on a
pl<=>");
```

Проверка на образец

Как и все проверки типов, она достаточно тривиальна:

Проверка на образец	
Нормальная форма	***Семантика***
is_pattern(Value)	*Истинно, если Value является образцом*

Выражение **is_pattern**(V) возвращает истинное значение, если результат *V* — произвольный образец (и ложное в противном случае).

Объекты и классы

В AWL присутствуют все основные средства для *объектно-ориентированного* программирования. Например, в языке присутствуют *объекты*, как вполне самостоятельный и независимый от других тип данных. (В отличие, например, от Perl — объекты в AWL никак не связаны со словарями.) Каждый объект является экземпляром определенного *класса* (который должен быть предварительно *декларирован*). Необязательными атрибутами классов являются *(пост)конструктор* и *деструктор*, возможность *наследования* от другого класса и *полиморфизм*, реализованный при посредстве *виртуальных* функторов.

Декларация класса

Каждый класс должен быть предварительно *декларирован*. Описание *(декларация)* класса во многом похоже на описание функтора (и это сходство не случайное). В самом общем случае, *декларация класса* имеет следующий вид:

```
!! [superclass] class (par1 par2 par3 … parN) :
   [memb1 memb2 memb3 … membN]
= constructor
~ destructor
# { virtual1 … virtualN }
{ decl1, decl2 … declN }
```

Прежде всего, декларация класса *всегда* начинается с *двух* (в отличие от декларации функтора) восклицательных знаков.
За ними (в перечисленном порядке) следуют:

- *необязательное* имя *суперкласса* (**superclass**) в квадратных скобках. (Если оно опущено, декларируемый класс не имеет суперкласса);
- *имя* определяемого класса (**class**),
- *необязательный* список *параметров класса* (*par1 … parN*),
- *необязательный* список *компонент класса* (*memb1 … membN*),
- *необязательные пост-конструктор* (*constructor*) и *деструктор* (*destructor*) класса;
- *необязательный* список *виртуальных методов* класса;
- *необязательный список локальных деклараций* класса (*decl1 … declN*).

Рассмотрим элементы описания класса по порядку.
Имя класса (**class**) принадлежит к пространству имен функторов. В этом пространстве оно должно быть уникально (т. е. не должно совпадать с именами пользовательских функторов или других классов). (При этом, оно может совпадать, например, с именем переменной.) Заметим, что само описание класса (так же, как и описание функтора) создает собственную, локальную область видимости (точнее, даже две области: для переменных класса, и для его методов).
Класс может иметь *суперкласс*, и в этом случае он *наследует* все атрибуты последнего. (Подробнее механизм наследования также рассмотрен ниже).
Список *компонент* класса содержит *переменные* (*они же компоненты*), принадлежащие каждому из объектов данного класса. В отличие от глобальных переменных, у каждой из компонент имеется ровно столько экземпляров, сколько существует объектов данного класса (т.е. в совокупности компоненты образуют набор уникальных данных для каждого экземпляра класса). Абсолютно то же относится к *параметрам класса* — разница состоит

лишь в том, что параметры класса автоматически инициализируются аргументами из списка при создании нового экземпляра класса (об этом подробнее ниже). Имена компонент и параметров принадлежат *области видимости класса* (т.е. они не мешают внешним переменным с теми же именами, но могут *временно* замещать их).
Дополнительно, класс может иметь *(пост)конструктор* и *деструктор*, обеспечивающие корректную инициализацию и уничтожение объектов данного класса (об этом также подробнее ниже).
Класс может декларировать ряд *виртуальных* функторов-методов (обычно переопределяемых в его потомках).
Наконец, *собственные декларации* класса — это, чаще всего, его методы (но иногда и другие классы), локализованные в описании декларируемого класса.
Заметим, что разделителем в этом списке всегда является *запятая* (не точка с запятой, как для большинства других деклараций!)

Создание объектов класса

После того, как новый класс декларирован, создать его новый экземпляр (т. е. *объект*) очень просто:

```
Object1 = class (arg1, arg2, … argN);
```

По сути дела, синтаксис создания объекта ничем не отличается от синтаксиса вызова функтора. Другими словами, декларируя новый класс, мы одновременно описываем и *функтор-конструктор* для экземпляров этого класса. При выполнении такого вызова создается новый объект класса **class**, и все его *компоненты-параметры* (*par1 ... parN*) инициализируются соответствующими аргументами из списка (*arg1 ... argN*). Собственно, в этом заключается единственная разница между параметрами и всеми прочими компонентами класса: первые инициализируются при создании нового экземпляра, а вторые — нет (т. е. им присваивается ()). После инициализации всех параметров, будет вызван *пост-конструктор*, если он присутствует (подробнее об этом позже). И в завершение, ссылка на созданный объект возвращается в качестве *результата вызова* (в данном случае, она присваивается переменной *Object1*).
Таким образом, процесс создания объекта очень похож на вызов функтора. Разница в том, что при вызове функтора он вычисляется и выдается результат; а при «вызове» класса — результатом является новый (инициализируемый параметрами) объект данного класса. Можно даже сказать, что класс — это функтор с приостановленным выполнением (т. к. все переданные ему при вызове значения сохраняются в виде экземпляра класса).
К аргументам класса применимо все то же, что и к аргументам «обычного» функтора. В частности, они также могут передаваться не по значению, а без вычисления (т. е. *лениво*), если перед именем аргумента стоит «@». Списковая семантика передачи аргументов в целом такая же, как и для вызова функтора. Более того: создание объекта (как и обращение к функтору) может быть не прямым, а *косвенным*, т. е. через вызов встроенного функтора **apply**:

```
Object2 = (!class) ! (arg1, arg2, … argN);
```

Поскольку класс принадлежит к тому же пространству имен, что и функторы — то для ссылки на него требуется запись вида !**class**. Однако, с полученным таким образом выражением можно делать все то что, что и со ссылкой на функторы. В частности, ее тоже можно вызвать через операцию «!» — получив в результате новый объект класса (в этом примере, *Object1* получит такое же содержимое, что и *Object2*). Используя ссылку на класс, в

процессе выполнения можно создавать экземпляры произвольного (заранее неизвестного) класса.
Как параметры, так и прочие компоненты класса могут отсутствовать в его декларации. В принципе, может даже отсутствовать и то, и другое — запись:

```
!! EmptyClass;
```

также формально является *законной* декларацией класса. (Для этого класса даже можно создавать экземпляры. Правда, они всегда будут пусты и не содержат каких-либо данных, поэтому на практической пользы от них будет немного.)

Квалифицированный доступ

При работе с классами имеется проблема *области видимости*. Важно всегда помнить, что поскольку все компоненты и внутренние декларации класса локализованы в нем — снаружи они по умолчанию недоступны. В общем случае, для того, чтобы обратиться к областям видимости классов, предусмотрена специальная языковая конструкция (*квалификатор*):

```
Class !! Expr
```

Здесь **Class** — имя класса, *Expr* — произвольное выражение (оно должно иметь замкнутый синтаксис). В остальном, выражение *Expr* может быть любым — но обычно оно содержит какие-то обращения к локальным декларациям класса **Class** (его компонентам, или локальным для него функторам). В результате, выражение *Expr* интерпретируется как вложенное в область видимости класса **Class**. Заметим, что эта языковая конструкция имеет эффект при *компиляции*. Именно поэтому здесь уместнее говорить об «*интерпретации*», а не об «*вычислении*» операнда *Expr*. По этой же причине, квалификации не соответствуют никакие средства периода выполнения (т. е. не существует никакой аналогичной «функторной» записи для выражения **Class** !! *Expr*).
Например, пусть имеется декларация класса:

```
!! MyClass (A B C) : [x y z] { … };
```

Здесь у *MyClass* имеется шесть компонент: его аргументы (*A*, *B*, *C*) и локальные переменные (*x*, *y*, *z*). *Внутри* декларации класса (мы все его внутренние описания сократили до «…») все перечисленное доступно просто по именам. Но *извне* декларации класса доступ внутрь должен быть *квалифицированным*: например, **MyClass**!!A, **MyClass**!!x и так далее. (Без этого, молча предполагается, что вы обращаетесь к *внешним* переменным *A*, *x* и т. п., не имеющим никакого отношения к классу **MyClass**, как и какому-либо еще классу.) Использовать такую громоздкую запись каждый раз для обращения к каждой компоненте класса (или его методу) было бы, конечно, очень неудобно. Однако, в качестве выражения *Expr* допускается что угодно (даже целый блок). Поэтому, можно написать, например:

```
MyClass!! (x*A + y*B + z*C)
```

- прямо указав на принадлежность все операндов этого выражения именно к области видимости **MyClass**. Проще говоря, рассматривайте класс **MyClass** как *черный ящик* (который по умолчанию *заперт*), а квалификатор **MyClass**!! как *ключ*, который позволяет этот ящик *временно* открыть. Заметим, что в языке (по крайней мере, в настоящее время) нет средств, позволяющих «открыть ящик» *постоянно* (то есть, сделать его «содержимое» видимым до конца программы). Это сделано в т.ч. ради безопасности: «загрязнение» области видимости нежелательно, увеличивая риск конфликта имен.

Как и любые выражения, квалификаторы могут быть *вложены* друг в друга. Допустимо написать что-нибудь вроде:

```
Class1 !! Class2 !! … ClassN !! Expr
```

что заставляет интерпретировать *Expr* в контексте всех классов **Class1** ... **ClassN**. При этом в *Expr* будут доступны все, что декларировано в перечисленных классах (как компоненты, так и методы). Конечно, при этом предполагается, что имена компонент (и/или методов) этих классов не конфликтуют. Если же конфликт имен возникает, он разрешается в пользу самых *внутренних* описаний: т. е. дублированные имена в **ClassN** переопределяют имена в **Class1** (но не наоборот). Но, вообще-то, таких ситуаций лучше избегать.
Также заметим, что первым операндом этой операции может быть *не только* имя класса: она равно применима к пользовательским функторам. Так, с помощью такой записи можно обратиться к локальной переменной (или параметру) функтора, из-за пределов области его видимости. (Необходимость в этом возникает не часто, но она может возникнуть.)
Наконец, квалификаторы применяются и для доступа к *встроенным пространствам имен* (примеры этого будут приведены далее).

Текущий экземпляр класса и его переопределение

Одним из фундаментальных понятий AWL является понятие *текущего экземпляра* для класса. Понятие это является довольно специфичным для AWL: в большинстве ОО-языков (таких, как C++ или Java) точных аналогов нет.
У *каждого* определенного пользователем класса имеется собственный *текущий экземпляр*, который является одним из его атрибутов (аналогично его компонентам). После декларирования нового класса его текущий экземпляр *отсутствует*. Однако, его можно (временно) определить, используя специальную операцию «.» (её функтор **with**).

(Пере)определение текущего экземпляра класса		
Синтаксис	***Нормальная форма***	***Семантика***
`Object . Expr`	**`with`**`(Object, Expr)`	*Выполнить выражение Expr (с объектом Object в качестве текущего экземпляра своего класса)*

Функтор **with** является *оболочкой* (т. е. функтором, который *временно* меняет среду выполнения). Если говорить более формально, то при его выполнении происходит следующее:

- Прежде всего, вычисляется значение первого операнда *Object*;
- если результат является объектом некого класса (обозначим его как **Class**), то…
- ... *предыдущий текущий экземпляр* класса **Class** запоминается;
- ... объект *Object* становится *новым текущим экземпляром* своего класса **Class**;
- ... вычисляется (или выполняется) *значение Expr*;
- ... восстанавливается *старый текущий экземпляр* класса **Class**;
- ... вычисленное значение *Expr* возвращается как результат.

Если первый операнд *не является* объектом, сообщается об ошибке (а результатом просто будет результат *Expr*).
Таким образом, текущий экземпляр класса можно изменить только *временно*. Однако, как

известно, “нет ничего более постоянного, чем временное”: поскольку выражение *Expr* может быть *произвольным* (в т. ч. блоком, или вызовом функтора), то возможно выполнить сколь угодно большой объем вычислений с *Object* в качестве текущего экземпляра для его **Class**. Наконец, важно отметить и то, что механизм, реализованный функтором **with**, полностью *реентрантен* — т.е. изменения текущего экземпляра для любого класса (или классов), могут быть *вложенными* на сколь угодно большую глубину.
Функторы, декларированные в классах, обычно принято называть его *методами*. На самом деле, в AWL это понятие определено несколько иначе, чем в большинстве языков. Например, в C++ или Java, “метод” отличается от «обычной» функции только тем, что у него есть дополнительный (неявный) аргумент — в виде ссылки на *объект класса*, к которому он применяется. Таким образом, понятие «текущего экземпляра» в этих языках тоже существует — но оно ограничено моментом вызова метода класса. В AWL дополнительные аргументы методов *отсутствуют* (они и не нужны), а «текущий экземпляр» класса определен *глобально*. (Если более строго — он определен там же, где определен рассматриваемый класс. Классы в AWL могут быть не только *глобальными*, но и *локальными*, например, для функторов!) Таким образом, в AWL применительно к объекту может быть вызван не только «метод класса», но и произвольный фрагмент кода, например.
Поскольку «методом» в AWL называется любой функтор, работающий с внутренним содержимым объекта класса — его можно декларировать и вне самого класса: В AWL можно легко создать внешний метод (**new_meth**) класса **Class** (что невозможно в C++ или Java):

```
! new_meth (Params) = Class !! Body;
```

Здесь квалификатор !! позволяет обращаться в *Body* ко «внутренностям» класса **Class** (совершенно так же, как и в собственных локальных функторах класса **Class**). Вызвать **new_meth** так же легко, как и любой другой локальный функтор из **Class**:

```
Object.new_meth (Args);
```

Операция “точка” похожа на свои аналоги в большинстве ОО-языков — но, на самом деле, ее семантика значительно шире. Как и в других языках, в AWL можно использовать эту операцию как для обращения к компоненте объекта:

```
` обращение к memb1, в объекте Object `
Object.Memb1;
```

так и для вызова функтора-метода:

```
` вызов func1, с параметрами (a,b,c) и с Object как текущим
объектом `
Object.func1 (a, b, c);
```

Однако, нижеследующие примеры перевести на другие языки будет труднее:

```
` сумма двух компонент объекта Object `
Object . (memb1 + memb2);

` произведение двух вызовов (оба вызваны с Object, как текущим
объектом) `
Object . (func1 (a, b, c) * func2 (d, e));
```

Наконец, напомним, что вторым операндом для **with** может быть и целый блок:

```
Object . { expr1; expr2; … exprN };
```

Здесь выполняется последовательность *expr1 ... exprN* с *Object* как текущим объектом своего класса. По своей сути, это очень похоже на оператор **with** в Паскале/Delphi, или на **inspect** в Simula. (В C++/Java аналогичных средств просто нет.)
Перечисленным, однако, все возможности далеко не исчерпываются. Так, в AWL нетрудно создать функтор, являющийся «методом» не одного, а *двух или более* классов:

```
! mutual_method_AB (Params) = ClassA !! ClassB !! Body;
```

Порядок квалификаторов здесь обычно несущественен (т. е. с тем же успехом можно было б написать **ClassB** !! **ClassA** !! *Body*). Также очевидно, что один из двух квалификаторов можно полностью опустить, если это описание локализовано внутри декларации для **ClassA** или **ClassB**. Здесь не показано, что именно происходит в *Body* — но предполагается некая *одновременная* работа, как с содержимым **ClassA**, так и с содержимым **ClassB**. Вызов «совместного метода» будет выполняться примерно так:

```
ObjectA. ObjectB. mutual_method_AB (Args);
```

или же так:

```
ObjectB. ObjectA. mutual_method_AB (Args);
```

(в данном случае, очевидно, порядок не играет существенной роли). Так же должно быть очевидно, что если *ObjectA*, или *ObjectB*, или они оба будут здесь опущены — то неявно будет использоваться *текущий* объект для данного класса (или для них обоих).
Теперь мы приведем некоторые более осмысленные примеры работы с классами. Для многих задач (например, компьютерной трехмерной графики) весьма полезны векторы в трехмерном пространстве. Мы определим класс **Vector3D**:

```
!! Vector3D (X Y Z) {
     ! length () = sqr (X*X + Y*Y + Z*Z)
};
```

```
! Vector3D.length:() [0] = sqr:add:(add:(mul:(Vector3D.X,
Vector3D.X), mul:(Vector3D.Y, Vector3D.Y)), mul:
(Vector3D.Z, Vector3D.Z)) => Vector3D.length:;

!! Vector3D:(X Y Z) {3} => Vector3D:;
```

В таком виде этот класс очень примитивен. Объект **Vector3D** содержит три координатных компоненты *X*, *Y*, *Z* (они же являются его параметрами, т. е. инициализируются при вызове), и внутренний функтор (если хотите, «метод») **length**, вычисляющий длину вектора по

стандартной формуле. Операция **with** здесь нигде явно не употребляется, т. е. функтор работает с *текущим экземпляром* **Vector3D**. Это значит, что если мы создадим несколько векторов:

```
Vec1 = Vector3D (10, 12, 15);
Vec2 = Vector3D (-3, -5, -11);
```

```
set:(Vec1, Vector3D:(10, 12, 15)) => Vector3D:{10, 12, 15};
set:(Vec2, Vector3D:(neg:3, neg:5, neg:11)) => Vector3D:{-
3, -5, -11};
```

то вычислить длины этих векторов можно будет следующим образом:

```
Vec1.Vector3D!!length ();
Vec2.Vector3D!!length ();
```

```
with:(Vec1, Vector3D.length:()) => 21.656408;
with:(Vec2, Vector3D.length:()) => 12.4499;
```

(Заметьте, что поскольку **length** является *внутренней* декларацией **Vector3D**, и мы обращаемся к ней *снаружи* — то используем квалификатор.) Однако, возможен и альтернативный подход к вычислению длины: с явной передачей вектора как параметра *внешнему* функтору (который назовем **Vec3D_length**).

```
! Vec3D_length(Vec) = Vector3D!! sqr (Vec.X*Vec.X +
Vec.Y*Vec.Y + Vec.Z*Vec.Z);
```

```
! Vec3D_length:(Vec) [1] = sqr:add:(add:(mul:(with:(Vec,
Vector3D.X), with:(Vec, Vector3D.X)), mul:(with:(Vec,
Vector3D.Y), with:(Vec, Vector3D.Y))), mul:(with:(Vec,
Vector3D.Z), with:(Vec, Vector3D.Z))) => length1:;
```

Ну а использовать **Vec3D_length** предполагается так:

```
Vec3D_length(Vec1);
Vec3D_length(Vec2);
```

```
Vec3D_length:Vec1 => 21.656408;
Vec3D_length:Vec2 => 12.4499;
```

Хотя эта реализация **Vec3D_length** корректна, но выполнена «в стиле C++/Java», т. е. не слишком изящна. На самом деле, поскольку мы все время обращаемся к *одному* объекту *Vec*, мы может «в стиле AWL» вынести его «за скобки» (т. е. соединить все операции **with** в одну-единственную). Тогда определение будет выглядеть так:

```
! Vec3D_length(Vec) = Vector3D!! sqr (Vec.(X*X + Y*Y + Z*Z));
```

или даже так:

```
! Vec3D_length(Vec) = Vector3D!! Vec.sqr (X*X + Y*Y + Z*Z);
```

(Опять-таки, оба варианта здесь совершенно эквивалентны.)

Рассмотрим теперь случай, когда функтор класса должен работать с *несколькими* экземплярами этого класса. Например, для векторов совершенно необходимы операции векторного *сложения* и *вычитания*. Реализовать их проще всего так (мы предполагаем, что этот код *добавляется* в саму декларацию класса **Vector3D**):

```
! add (VA VB) = Vector3D(VA.X + VB.X, VA.Y + VB.Y, VA.Z +
VB.Z),
! sub (VA VB) = Vector3D(VA.X - VB.X, VA.Y - VB.Y, VA.Z -
VB.Z)
```

```
! Vector3D.add:(VA VB) [2] = .Vector3D:(add:(with:(VA,
Vector3D.X), with:(VB, Vector3D.X)), add:(with:(VA,
Vector3D.Y), with:(VB, Vector3D.Y)), add:(with:(VA,
Vector3D.Z), with:(VB, Vector3D.Z))) => Vector3D.add:;
! Vector3D.sub:(VA VB) [2] = .Vector3D:(sub:(with:(VA,
Vector3D.X), with:(VB, Vector3D.X)), sub:(with:(VA,
Vector3D.Y), with:(VB, Vector3D.Y)), sub:(with:(VA,
Vector3D.Z), with:(VB, Vector3D.Z))) => Vector3D.sub:;
```

Как и выше, это описание вполне корректно — но оно *неоптимально*. В данном случае, «вынести за скобки» *оба* операнда, очевидно, не получится (*текущий* экземпляр у класса в каждый момент времени может быть только один)! Однако, вполне можно «вынести» любой из них, по вашему выбору. Таким образом, **add** можно переписать или так:

```
! add(VA VB) = VA.Vector3D(X + VB.X, Y + VB.Y, Z + VB.Z),
```

или же так:

```
! add(VA VB) = VB.Vector3D (VA.X + X, VA.Y + Y, VA.Z + Z),
```

По своей функиональности, оба варианта вполне равнозначны. (Аналогично, конечно, можно поступить и с **sub**.)
Каким бы из перечисленных способов эти функторы не были определены — работа с ними вполне тривиальна:

```
Vector3D!! add (Vec1, Vec2);
Vector3D!! sub (Vec2, Vec1);
```

```
Vector3D.add:(Vec1, Vec2) => Vector3D:{7, 7, 4};
Vector3D.sub:(Vec2, Vec1) => Vector3D:{-13, -17, -26};
```

Разумеется, операции для сложения и вычитания векторов могут быть легко реализованы и как *внешние* для **Vector3D** функторы (например, под названием **Vec3D_add** и **Vec3D_sub**). Их реализацию в предложенном виде оставим для самостоятельного упражнения.
В завершение, еще раз подчеркнем различия и сходство между операциями «!!» и «.»:
- первая операция работает при *компиляции* кода, а вторая — при его *выполнении*;
- первая операция имеет первым операндом *имя класса*, а вторая — *объект*.

Сходство же между ними заключается в том, что обе они имеют *локальный эффект* (т. е. действуют только на свой второй операнд *Expr*). Первая операция указывает (при компиляции), что мы обращаемся к некоторому классу (**Class**), а вторая — что мы работаем с конкретным объектом (экземпляром) *Object* заданного класса.

Атрибуты классов и атрибуты объектов

Напомним, что поскольку каждый класс является специальной разновидностью функтора, то *ссылки на классы* являются подмножеством ссылок на функторы. Каждый объект является экземпляром некоторого класса — и с помощью этого функтора можно узнать, какого именно:

Узнать класс объекта	
Нормальная форма	***Семантика***
class_of(Object)	*Возвращает ссылку на класс, экземпляром которого является Object*

Так, для примеров из предыдущей главы:

```
class_of (Vec1);
class_of (Vec2);
```

```
class_of:Vec1 => Vector3D:;
class_of:Vec2 => Vector3D:;
```

Если результатом вычисления операнда *Object* будет не объект, функтор сообщает об ошибке, и возвращает (). А в реальной программе (для безопасности) иногда полезно проверить, действительно ли некоторое выражение возвращает *объект*:

Проверка на объект	
Нормальная форма	***Семантика***
is_object(Expr)	*Истинно, если Expr является объектом.*

Этот предикат возвращает *истинное* значение, если результатом *Expr* является объект (произвольного класса), и *ложное* — если результатом является что-либо еще.

Рассмотрим теперь операции, определяющие атрибуты не объектов, а самих *классов*. Прежде всего, для любого класса можно узнать его *текущий экземпляр*:

Текущий экземпляр класса	
Нормальная форма	***Семантика***
self(Class)	*Объект — текущий экземпляр класса Class.*

Единственный операнд функтора — ссылка на класс, а его результат — текущий экземпляр данного класса (или (), если текущий экземпляр у него *отсутствует*). Например:

```
self (!Vector3D);
```

позволяет узнать текущий экземпляр класса **Vector3D**. Это примерно то же самое, что в C++ или Java обеспечивает **this** (но операнд, ссылка на класс, должен быть явным). Функтор сообщает об ошибке, если операнд не является ссылкой на какой-либо класс. Заметим, что хотя всякая ссылка на класс является ссылкой на функтор, но обратное — уже неверно:

Проверка на класс	
Нормальная форма	***Семантика***
is_class(Func)	*Истинно, если Func является ссылкой на класс.*

Этот предикат позволяет отличить ссылки на классы от прочих функторных ссылок: для первых он возвращает истинное значение, а для всех прочих (например ссылок на встроенные или пользовательские функторы) — ложное.

«Отключение» текущего экземпляра

Как мы уже говорили раньше, по умолчанию (в т.ч. и немедленно после декларирования нового класса) его текущий экземпляр является *отсутствующим*, но с помощью операции «.» (функтор **with**) его можно временно определить (или же *пере*определить). Применять эту операцию можно совершенно независимо от того, был у класса текущий экземпляр, или его не было: в первом случае, он *временно* сохраняется; а во втором — он *временно* становится определенным (но по завершении операции, вновь объявляется отсутствующим). Однако, иногда (хотя и редко) бывает нужно временно *«разопределить»* текущий экземпляр для некоторого класса (т. е. сделать его «отсутствующим», восстановив ситуацию «при рождении» класса). Функтор **with** для этого использовать *невозможно*: он всегда требует первый операнд в виде *реального* экземпляра некоторого класса. Поэтому, для этого предусмотрен отдельный функтор **without**:

«Отмена» текущего экземпляра класса	
Нормальная форма	***Семантика***
without(Class, Expr)	*Выполнить выражение Expr, с отстутствующим текущим экземпляром класса Class*

Важно отметить, что (в отличие от **with**) для **without** первым операндом является не *объект*, а *ссылка на класс*. Однако (точно так же, как и **with**) **without** действует сугубо *локально*: т. е. выполняет свой второй операнд *Expr* с *отсутствующим* текущим экземпляром для класса **Class**. То есть, если у **Class** *имелся* определенный текущий экземпляр — он будет бережно сохранен (и после выполнения операции восстановлен). (Если же его и так не было — операция, фактически, ничего не осмысленного делает — кроме вычисления результата *Expr*, разумеется.) Но, в любом случае, результатом всей операции — будет результат *Expr*.

Конструкторы и деструкторы

Дополнительными (необязательными) атрибутами класса в AWL могут быть его *конструктор* и/или *деструктор*. Их основная задача — обеспечить корректную *инициализацию* и/или *уничтожение* экземпляров данного класса.

Подход к этой проблеме в AWL сильно отличается от принятого в C++ или Java. В отличие от этих языков, конструкторы и деструкторы AWL не являются специализированными методами — это просто выражения (которые, как всегда, могут быть и произвольными блоками кода). Строго говоря, «конструктором класса» (имеющим параметры) в AWL является сам класс (и он имеется только один). То, что мы обсуждаем здесь, технически грамотнее именовать *пост-конструктором* (т. к. он вызывается уже после того, как экземпляр класса создан) — но это звучит слишком громоздко.

Функции пост-конструктора могут быть такими:

- инициализировать те внутренние компоненты класса, которые не являются его аргументами (и потому не инициализированы явно при вызове);
- компоненты, инициализированные как параметры, тоже иногда нужно подкорректировать (например, чтобы они соответствовали требованиям внутренней целостности объекта);
- наконец, конструктор может работать с *внешними* по отношению к классу данными (или с внешними по отношению к программе ресурсами — например, файлами или окнами графической системы).

У *деструктора*, очевидно, совершенно противоположные задачи. Деструктор вызывается (вычисляется) непосредственно перед уничтожением данного объекта класса. Так, если класс задействовал какие-то внешние ресурсы, деструктор может их освободить.

Приведем пример весьма тривиального класса — экземпляры которого, однако, умеют информировать о собственном “рождении” и “смерти”.

```
!! traced_class (A B)
= { <: ['[created: traced_class (' A ',' B ')]\n' ] }
~ { <: ['[destroyed: traced_class (' A ',' B ')]\n'] };
```

Поэкспериментировав с объектами **traced_class**, можно убедиться в том, что их создание и уничтожение происходят во вполне определенные моменты времени. (В AWL применяется *детерминированное* управление памятью, в отличие от Java и некоторых других языков).

```
tr_a = traced_class (101, 303);
tr_b = traced_class ("aabb", "ddee");
```

```
[created: traced_class (101,303)]
set:(tr_a, traced_class:(101, 303)) => traced_class:{101,
303};
[created: traced_class (aabb,ddee)]
set:(tr_b, traced_class:("aabb", "ddee")) => traced_class:
{"aabb", "ddee"};
```

```
tr_a = tr_b = ();
```

```
[destroyed: traced_class (aabb,ddee)]
[destroyed: traced_class (101,303)]
set:(tr_a, set:(tr_b, ())) => ();
```

В целом, здесь нет особых сюрпризов. Обычно, объект прекращает свое существование, когда последняя ссылка на него исчезает (именно в этот момент вызывается его деструктор, если он определен).

То, что у класса есть только один «официальный» конструктор, может показаться неудобным. Однако, на самом деле, никто не мешает определить для класса произвольное количество «неофициальных» конструкторов (обычно, это делается внутри класса — хотя, в принципе, можно определить их и «снаружи»). Просто (в отличие от официального) эти конструкторы будут именованными. В отличие от C или C++, в AWL в принципе не существует *перегруженных* функторов: имя каждого функтора (в своей области действия) должно быть уникальным. Соответственно, и конструктор класса тоже не может быть «перегруженным». Поэтому, если предусмотрено множество способов создания нового экземпляра класса, то каждый из них может быть реализован функтором, имеющим свое уникальное имя. «Официальный» конструктор всегда только один, и объявляется автоматически, «в нагрузку» к определению класса.

Производные классы

До сих пор, мы (для простоты) намеренно не рассматривали вопрос о *наследовании*. Все рассматриваемые нами классы были *оригинальными*, т. е. не имели явного «предка». В отличие от многих ОО-языков, в AWL не существует «метаклассов», или абстрактных классов с названиями вроде **Object**, от которых происходят все остальные. Если класс не имеет явно декларированного «предка» — значит, его создание происходит «с чистого листа».

Однако, класс также может быть и *производным* от любого другого. Если в декларации перед именем класса стоит *имя суперкласса* (в квадратных скобках) — значит, декларируемый класс является *потомком* указанного суперкласса. При этом автоматически наследуются все его атрибуты, в т.ч. его компоненты (параметрические и обычные) и его методы. За счет этого обычно наследуется и большая часть *функциональности*, заложенной в его суперкласс. При этом, производный класс, естественно, может иметь и *собственный* набор компонент и локальных функторов (дополняющих то, что унаследовано от суперкласса). Заметим, что в AWL (по крайней мере — на данный момент) наследование классов исключительно *линейное*: суперкласс у класса может быть только один.

Приведем примеры очень простой иерархии классов:

```
!! ClassA (a b c);

!! [ClassA] ClassAB (m n);

!! [ClassAB] ClassABC (x y z);
```

Здесь **ClassAB** является потомком **ClassA**, а **ClassABC —** потомком **ClassAB**. В дополнение к собственным параметрам, у **ClassAB** имеются параметры (*a*, *b*, *c*); а у **ClassABC** — еще и (*m*, *n*). (О корректном способе инициализации этих параметров мы расскажем немного ниже.)

Как и к любому классу, для доступа к внутренностям суперкласса *извне* нужны *квалификаторы*. Производный класс отличается от оригинального тем, что при квалифицированном обращении к нему — автоматически становятся доступными также пространства имен всех его суперклассов. Таким образом, с практической точки зрения имена компонент для всех предков класса можно считать частью его пространства имен (так же, как и имена всех функторов, декларированных в предках). На самом деле, конечно, у каждого класса собственное пространство имен — но пространства имен суперклассов автоматически включаются в пространство имен производного класса. Например, выражение:

```
ClassAB!!Expr;
```

имеет тот же эффект, что и:

```
ClassA!!ClassAB!!Expr;
```

Аналогично, выражение:

```
ClassABC!!Expr;
```

равносильно:

```
ClassA!!ClassAB!!ClassABC!!Expr;
```

Более того: компоненты (и методы) производного класса могут иметь то же имя, что и у суперкласса — и в этом случае они их временно замещают. Однако (хоть это и допустимо) злоупотреблять этим очень не рекомендуется (конфликты имен вообще являются плохой идеей). (Если же нужно переопределить в производном классе не просто имя, а *функциональность* какого-то метода — для этого правильно будет использовать *виртуальные методы*, детально описанные в следующей главе.)

Операция «точка» (т. е. **with**) работает с объектами производных классов так же, как и с оригинальными. Разница в том, что при использовании этой операции объект автоматически становится текущим экземпляром не только для своего класса — но и для *всех его суперклассов*! Конечно, эти изменения являются временными: все старые экземпляры суперклассов сохраняются при входе в **with** и восстанавливаются при выходе из него. Например, если *ObjectAB* является экземпляром **ClassAB —** то в выражении *ObjectAB.Expr* этот объект временно становится текущим экземпляром не только для **ClassAB**, но и для **ClassA**. Аналогично, если *ObjectABC* является экземпляром **ClassABC —** то в выражении *ObjectABC.Expr* он становится текущим для классов **ClassABC**, **ClassAB** и **ClassA**.

Объект производного класса тоже может быть инициализирован списком аргументов. При этом, однако, возникает вопрос: а каким образом инициализируется его суперкласс? Ответ простой: *первым аргументом в списке* всегда является инициализатор для суперкласса (в общем случае, это также список), а все дальнейшие аргументы передаются производному классу. Это правило применяется рекурсивно: т. е., для нескольких уровней наследования, первый аргумент — передается суперклассу, первый аргумент первого аргумента — суперклассу суперкласса и т.п. Если суперклассу не требуется явный список параметров, можно задать вместо него (). Заметим, что даже когда суперкласс вообще не имеет параметров — аргумент для него (пусть даже и пустой) *должен* присутствовать!

```
ObjectA = ClassA (10, 20, 30);

ObjectAB = ClassAB ((30, 40, 50), 11.1, 12.2);

ObjectABC = ClassABC (((60, 70, 80), 13.3, 14.4), "x", "y",
"z");
```

Впрочем, все сказанное относится к «официальному конструктору» производного класса. Если пользоваться им неудобно, всегда можно определить и «неофициальные» конструкторы.

Конечно же, производный класс может иметь также собственные *конструкторы* и *деструкторы*. Как они работают, должно быть уже примерно ясно. Именно, если у суперклассов имеются конструкторы, все они вызываются для каждого создаваемого объекта. (При этом, *самый внешний* конструктор вызывается *первым*, а *самый внутренний — последним* (но каждый из конструкторов вызывается *после инициализации* параметров, соответствующих данному уровню иерархии). Если у производного класса есть деструкторы, то все они вызываются при уничтожении объекта данного класса. Порядок их вызова противоположен порядку вызова конструкторов: *первым* вызывается деструктор для самого *внутреннего* класса, а *последним* — для самого *внешнего*.

От механизма наследования классов было бы не слишком много пользы, если б производные классы можно было только *дополнять* новыми методами — но при этом не было возможности *переопределять* методы своих предков. Возможность для этого предоставляют *виртуальные методы*.

Виртуальные методы

Этот механизм обеспечивает *полиморфный* вызов функторов, т. е. способ для производного класса создать альтернативную версию одного или нескольких функторов-методов. Такие функторы мы будем называть *виртуальными*. Для тех, кто знаком с C++ или Java, такая идея в целом знакома. Рассмотрим отличия.

Прежде всего, в AWL как декларация, так и реализация виртуальных функторов имеют специальный синтаксис. Для обычных функторов (как глобальных, так и методов класса) декларация одновременно является и реализацией. Однако, каждый виртуальный метод декларируется один раз, но реализован может быть многократно: и декларирующий его класс, и каждый из его потомков может иметь собственную версию для него.

Декларация виртуальных методов является специальной (необязательной) частью описания класса (которая идет после конструктора и деструктора, но до собственно тела класса). Синтаксис этой декларации очень прост: она начинается с символа «#», за которым идет (в фигурных скобках) список имен декларируемых виртуальных методов (разделенных пробелами). Например:

```
!! MySuperClass
# { virtual1 virtual2 }
{
…
};
```

Здесь класс **MySuperClass** объявляет два виртуальных функтора (**virtual1** и **virtual2**), которые будут доступны как в нем, так и во всех производных от него классах. Класс, в котором виртуальный функтор декларирован, мы будем далее называть его *оригинатором*. (Очевидно, что у каждого виртуального метода оригинатор только один!) Декларация виртуальных функторов ничего не говорит об их параметрах или выполнении: все это определяется исключительно его реализациями. Фактически, сам виртуальный функтор — это только имя (и некий внутренний индекс, присваиваемый ему автоматически).

Но, конечно, чтобы эти функторы делали что-то осмысленное, надо еще их *реализовать*. Как класс-оригинатор, так и любой производный от него класс может реализовать любые из своих виртуальных методов (разумеется, каждый метод для каждого класса может быть реализован только однократно). *Реализация* виртуального функтора отличается от декларации обычного только тем, что перед его именем тоже ставится «#». Все остальные компоненты описания (список параметров, список локальных переменных, тело функтора) — такие же, как всегда (и работают точно так же).

Например, в теле класса **MySuperClass** можно написать следующее:

```
! #virtual1 = <: "Greetings from SuperClass!!!\n"
```

Наличие # перед именем функтора важно: без него это будет декларацией отдельного функтора, никак не связанного уже объявленными виртуальными. Если у этого класса имеются потомки, то **virtual1** в любом из них можно переопределить:

```
!! [MySuperClass] MySubClass {
  ! #virtual1 = <: "Greetings from SubClass!\n"
};
```

Сам механизм виртуальных вызовов работает следующим образом: при вызове **virtual1** вызывается та реализация этого метода, которая активна для *текущего экземпляра* класса

MySuperClass (которым могут быть, разумеется, и все производные от него классы). Таким образом, **MySuperClass**().**virtual1**() выведет строку “Greetings from SuperClass!”, а **MySubClass**().**virtual1**() соответственно, выведет “Greetings from SubClass!”. Конечно, в потомках класса **MySubClass** метод **virtual1** также может быть переопределен (например, может выводить что-то еще). Если виртуальный метод *не был* явно переопределен в субклассе, заимствуется его версия из суперкласса. Виртуальный метод вообще может быть оставлен *без определения* (как, в нашем примере, **virtual2**). Вызов не имеющего определения виртуального метода ничего не делает, возвращая (). (Это не вполне разумно: в будущем, скорее всего, это будет вызывать явное сообщение об ошибке.)

Процесс выбора текущей версии для виртуального метода называется его *девиртуализацией*. Обычно она имеет место только *при его вызове*. Заметим, что виртуальный функтор тоже может быть вызван не только напрямую, но и *через ссылку*. Однако, и в этом случае, его девиртуализация происходит не в момент получения ссылки на него, а только в момент его вызова. Другими словами, виртуальный метод всегда представляет собой своего рода *интерфейс* для доступа ко множеству «физических» методов.

Любой из производных классов также может стать *оригинатором* для собственных виртуальных методов. Они также могут иметь различные реализации как в нем, так и в его подклассах. Заметим, что в AWL нет возможности *запретить* (*sealed method*) переопределение виртуального метода для потомков какого-либо класса.

Приведем теперь более содержательный пример полиморфного поведения объектов. Пусть мы хотим работать с *абстракциями*, представляющими собой простые геометрические фигуры. Для этого будет логично создать *абстрактный* класс **Shape**:

```
` Абстрактный геометрический объект `
!! Shape
     # {
     put       ` вывести информацию о фигуре `
     perimeter ` вычислить периметр `
     area      ` вычислить площадь `
     };
```

Он очень тривиален. Вся функциональность класса заложена в его виртуальные функторы (все без аргументов): **put** позволяет выводить информацию о фигуре (в стандартный вывод), **perimeter** вычисляет ее периметр, а **area —** ее площадь.

Теперь нетрудно создать набор конкретных геометрических фигур (потомков класса **Shape**):

```
` Прямоугольник (со сторонами A, B) `
!! [Shape] Rectangle (A B) {
     ! #put = <: ['Rectange (' A ' * ' B ')'],
     ! #perimeter = 2*(A + B),
     ! #area = A*B
     };
```

```
` Квадрат (со стороной S) `
!! [Shape] Square (S) {
     ! #put = <: ['Square [' S ']'],
     ! #perimeter = 4*S,
     ! #area = S*S
     };
```

```
` Круг (с радиусом R) `
!! [Shape] Circle (R) {
     ! #put = <: ['Circle [' R ']'],
     ! #perimeter = 2*pi(R),
     ! #area = pi(R*R)
     };
```

```
` Треугольник (со сторонами A, B, C) `
!! [Shape] Triangle (A B C) {
     ! #put = <: ['Triangle (' A ',' B ',' C ')'],
     ! #perimeter = A + B + C,
     ! #area : [p] = {
          `(...формула Герона...)`
          p = (A + B + C) / 2;
          sqr (p*(p - A)*(p - B)*(p - C))
          }
     };
```

Теперь уже легко можно создать список из конкретных геометрических фигур.

```
Figures = (
     Square ((), 5),
     Rectangle ((), 6, 9),
     Circle ((), 10),
     Triangle ((), 5, 12, 13)
     );
```

```
set:(Figures, Square:((), 5), Rectangle:((), 6, 9), Circle:
((), 10), Triangle:((), 5, 12, 13)) => (Square:{Shape:{}->
5}, Rectangle:{Shape:{}-> 6, 9}, Circle:{Shape:{}-> 10},
Triangle:{Shape:{}-> 5, 12, 13});
```

Заметьте, что первый аргумент в списке инициализаторов для каждой фигуры — (), т. к. он требуется для суперкласса **Shape** (даже несмотря на то, что этот суперкласс не имеет явных параметров).
Работать с созданным списком фигур можно так:

```
Shape!!l_loop (fig, Figures, fig.{
     put ();
     <: ": ";
     <: ("периметр = ", perimeter(), "; ");
     <: ("площадь = ", area());
     <: "\n";
     });
```

```
Square [5]: периметр = 20; площадь = 25
Rectange (6 * 9): периметр = 30; площадь = 54
Circle [10]: периметр = 62.831853; площадь = 314.15927
Triangle (5,12,13): периметр = 30; площадь = 30.
```

В заключение, с виртуализацией также связано несколько полезных встроенных предикатов. Прежде всего, нетрудно проверить, имеем ли мы дело с виртуальным методом:

Проверка на виртуальный метод	
Нормальная форма	***Семантика***
`is_virtual``(Func)`	*Истинно, если Func является ссылкой на виртуальный метод.*

Этот предикат истинен, только если *Func* является ссылкой на виртуальный метод какого-либо класса. В этом случае, также нетрудно выяснить, какого именно:

Оригинатор виртуального метода	
Нормальная форма	***Семантика***
`originator``(Func)`	*Возвращает ссылку на класс-оригинатор для виртуального метода Func.*

(Средства для «явной девиртуализации») здесь не описаны.

Файловый ввод-вывод

Как уже говорилось выше, в AWL-программе всегда присутствуют три *потока*: ввода, вывода и вывода ошибок. Первый из них всегда доступен для чтения, остальные два — всегда доступны для записи. Обычно, все эти три потока для AWL-программы связаны с консолью, но средствами операционной системы это (как правило) можно изменить, заставив вашу программу читать и/или записывать данные, например, в дисковый файл.
Однако, помимо этого, в AWL, конечно, присутствуют и *явные* средства для работы с заданными файлами.

Открытие файла

Чтобы начать работать с файлом на диске (или другом носителе) — его надо, прежде всего, открыть. Эта операция создает и возвращает *поток ввода/вывода*, связанный с указанным файлом.

Открытие файла	
Нормальная форма	***Семантика***
`f_open``(FileName[, Mode])`	*Открывает файл FileName в режиме Mode (режиме чтения по умолчанию)*
`f_create``(FileName[, Mode])`	*Открывает файл FileName в режиме Mode (режиме создания/записи по умолчанию)*

Фактически — это два разных имени для одного и того же функтора. Они различаются, лишь когда параметр *Mode* отсутствует (когда он задан, режимом открытия файла управляет исключительно он). Этот параметр является комбинацией из следующих числовых флагов (в стандартной библиотеке им присвоены символические имена):

O_RDONLY (**0**)	Открыть файл только для чтения.
O_WRONLY (**1**)	Открыть файл только для записи.
O_RDWR (**2**)	Открыть файл одновременно для чтения и записи.
O_APPEND (**8**)	Открыть файл в режиме добавления. (Дополнительно к режиму записи — гарантирует, что каждая операция записи будет осуществляться точно в конец файла. Имеет смысл, если в файл пишут несколько процессов сразу.)
O_CREAT (**\x100**)	Режим создания файла. (Дополнительно к режиму записи — если файла не было, он автоматически создается.)
O_TRUNC (**\x200**)	Режим усечения файла. (Дополнительно к режиму записи — если файл уже был, он укорачивается до нулевой длины. Все данные при этом теряются!)

O_EXCL (**\x400**)	Исключительно создание. (Дополнительно к режиму записи — если файл уже был, его открытие завершается неудачно.)

Точная интерпретация имени файла *FileName* определяется операционной системой. (В Windows, MacOS и Unix достаточно разные соглашения относительно имен файлов.)
Открытие файла, конечно, не обязательно всегда завершается успешно. Если файл не существует (и флаги *O_CREAT* или *O_EXCL* отсутствует), открыть его не удастся. Даже если он существует, но вы не имеете права открыть его в заданном режиме (например, писать в него, или читать) — операция также завершится неудачно. При неудаче, функторы возвращают (), в противном случае — возвращают новый поток.

Чтение из потока

Для выполнения любых операций чтения из потока — он должен допускать чтение. Стандартный ввод (**f_in** ()) доступен для чтения всегда. Если же поток создан в результате открытия файла — он должен открываться в режиме *O_RDONLY* или *O_RDWR*.
Стандартную операцию чтения из файла (**f_get**) мы уже рассматривали. У нее есть альтернативы:

Чтение строки из потока	
Нормальная форма	***Семантика***
f_getline(Stream)	*Читает и возвращает строку из входного потока Stream.*

Этот функтор работает так же, как и **f_get** — но, в отличие от него, *не является* мутатором, и прочитанную строку просто возвращает как результат. (Как и для **f_get**, символ конца строки в результат НЕ включен.) Если достигнут конец файла, возвращается ().

Чтение блока данных из потока	
Нормальная форма	***Семантика***
f_read(Stream, Count)	*Прочитать строку в Count символов из входного потока Stream.*

В отличие от **f_getline**, **f_read** пытается прочитать непрерывный блок данных заданной длины (полностью игнорируя обычную разбивку файла на строки). Результат возвращается в виде строки, длины *Count* (она может быть и короче, если конец файла был достигнут преждевременно).

Чтение символа из потока	
Нормальная форма	***Семантика***
f_getc(Stream)	*Возвращает код следующего символа из входного потока Stream.*

Этот функтор возвращает код следующего символа (число, а не строку). Если достигнут

конец файла, возвращается ().
Во всех перечисленных функторах аргумент *Stream* может быть пустым: тогда подразумевается *стандартный ввод*. (Однако, см. далее **with_input**.)

Запись в поток

Для выполнения любых операций записи в поток — он должен допускать запись. Стандартный вывод (**f_out** ()) и вывод ошибок (**f_err** ()) доступны для записи всегда. Если же поток создан в результате открытия файла — он должен открываться в режиме *O_WRONLY* или *O_RDWR*.
Операцию записи в файл (**f_put**) мы уже рассматривали. Существуют и альтернативы:

Запись символа в поток	
Нормальная форма	***Семантика***
f_putc(Stream, Code)	*Записывает символ с кодом Code в поток Stream.*

Эта операция выводит символ с кодом *Code* (число, не строка!) в поток *Stream* (возвращая *Code* в качестве результата).

Запись строки в поток	
Нормальная форма	***Семантика***
f_putline(Stream, String)	*Записывает строку String в поток Stream.*

Эта операция выводит строку *String* в поток *Stream* (возвращая *String* в качестве результата).

Во всех перечисленных функторах аргумент *Stream* может быть пустым: в этом случае подразумевается *стандартный вывод*. (Однако, см. далее **with_output**.)

Переопределение стандартных потоков

Стандартные потоки для ввода и вывода данных обычно определяются операционной системой перед запуском программы. Однако, в процессе ее выполнения, их тоже можно (временно) переопределить.

Переопределение стандартного ввода/вывода	
Нормальная форма	***Семантика***
with_input(Stream, @Body)	*Определив входной поток Stream, как стандартный ввод, выполнить Body.*
with_output(Stream, @Body)	*Определив выходной поток Stream, как стандартный вывод, выполнить Body.*

Функтор **with_input** временно делает аргумент *Stream* стандартным *входным* потоком (т. е. потоком по умолчанию для функторов **f_get**, **f_getc**, **f_getline**, **f_read**). Функтор **with_output** временно делает *Stream* стандартным *выходным* потоком (т. е. потоком по умолчанию для функторов **f_put**, **f_putc**, **f_putline**). Обе операции действуют *локально* (т. е. на время выполнения аргумента *Body*): перед выполнением *Body* старое значение потока сохраняется, а после — восстанавливается. Результатом вызова является результат *Body*.

Например, перенаправить вывод программы в стандартный вывод ошибок можно так:

```
with_output (f_err ()) :: { … };
```

При выполнении кода в блоке {…}, каждая операция вывода (для которой выходной поток не будет специфицирован *явно*) — будет обращаться к потоку вывода диагностики (**f_err** ()) вместо стандартного вывода.
Заметим, что на значения, возвращаемые функторами **f_in** () и **f_out** (), эти операции *не влияют* (они всегда соответствуют глобальным для программы стандартным потокам ввода и вывода). Поэтому, используя:

```
with_input (f_in ()) :: { … };
```

или:

```
with_output (f_out ()) :: { … };
```

можно на время выполнения блока {…} вернуться к глобальным (для программы) стандартным потокам.

Позиционирование в потоке ввода/вывода

Обычно все операции чтения и записи в потоке выполняются *последовательно* (т. е. каждая из них начинает выполняться там, где завершилась предыдущая). Однако, есть и операции, позволяющие *перемещать* место следующего чтения или записи (абсолютно, или относительно текущей позиции). (Сразу заметим, что все эти операции поддерживаются не для всех потоков — обычно, только для связанных с дисковыми файлами.)

Абсолютное позиционирование потока	
Нормальная форма	***Семантика***
f_seek(Stream, Pos)	*Установить позицию чтения/записи потока Stream на Pos (относительно начала или конца).*

Значение аргумента *Pos* должно быть целым числом. Если он положителен, устанавливается позиция относительно начала (0 соответствует *началу* потока). Если он отрицателен, устанавливается позиция относительно конца (-1 соответствует *концу* потока). Прежнее положение указателя полностью игнорируется. Результатом является новая позиция (относительно начала потока), или -1 (если переместить указатель не удалось).

Относительное позиционирование потока	
Нормальная форма	***Семантика***
f_skip(Stream, Offset)	*Сдвинуть позицию чтения/записи потока Stream на Offset (вперед или назад).*

Эта операция устанавливает новую позицию *относительно текущей*, смещая ее *назад* (если *Offset* отрицательно), или *вперед* (если *Offset* положительно). Результатом (как и для **f_seek**) является новая позиция (относительно начала), или -1 (при неудаче). Если *Offset* равно 0, ничего не меняется.

Текущая позиция потока	
Нормальная форма	***Семантика***
f_tell(Stream)	*Текущая позиция потока Stream.*

Результатом является *текущая позиция* потока *Stream* (или -1, если поток не поддерживает позиционирование). (То же значение возвращает **f_skip** (*Stream*, 0).)

Длина файла, связанного с потоком

Если поток ввода/вывода связан с файлом, можно узнать его текущую длину (а для выходного потока — и изменить ее).

Длина потока	
Нормальная форма	***Семантика***
f_get_length(Stream)	*Вернуть текущую длину потока Stream.*

Операция возвращает текущую длину потока (то же значение, которое вернет **f_seek** (*Stream*, -1), но указатель при этом не перемещается). Результатом является -1, если длина для потока не определена.

Изменение длины потока/файла	
Нормальная форма	***Семантика***
f_set_length(Stream, Len)	*Изменяет длину потока Stream на Len.*

Результатом является *Len* (при успехе), или -1 (если изменить длину файла не удалось). Осторожно, операция *крайне деструктивна*: когда файл укорачивается, часть его данных уничтожается бесследно!

Закрытие потока

Для закрытия потока предусмотрена отдельная операция:

Закрытие потока	
Нормальная форма	***Семантика***
f_close(Stream)	*Закрывает поток Stream.*

Операция закрывает ранее открытый поток *Stream*, прекращая любые связанные с ним операции ввода/вывода. Результатом является 0 (при успешном закрытии), или -1 (при ошибке). После закрытия, любые операции со *Stream* являются некорректными (и вызывают сообщение об ошибке).

Операция часто избыточна, т. к. когда все ссылки на поток исчезают — он закрывается автоматически. Однако, явное закрытие потока является хорошим тоном (к тому же, позволяет удостовериться, что поток действительно был закрыт корректно).

Закрыть стандартные потоки программы (**f_in**(), **f_out**(), **f_err**()) нельзя! Попытка сделать это вызывает сообщение об ошибке.

Проверка на поток

Большинство операций, работающих с потоками, сообщают об *ошибке*, если их аргумент не является потоком. Конечно, как всегда, для дополнительной безопасности это можно проверить явно:

Проверка на поток	
Нормальная форма	***Семантика***
is_stream(Expr)	*Аргумент Expr является потоком?*

Предикат **is_stream** истинен, если результатом *Expr* является поток, и ложен в противном случае.

Файловая система

Рассмотрим прочие системные функторы, имеющие отношение к файлам и файловой системе.

Проверка доступности файла

Даже не открывая файл, можно проверить возможности доступа к нему. (Операция в основном функционально аналогична системному вызову **access**(2) в UNIX.)

Проверка доступа к файлу	
Нормальная форма	***Семантика***
`f_access``(FileName, Access)`	*Проверяет, доступен ли файл FileName в режиме доступа Access.*

Параметр *FileName* задает имя проверяемого файла, параметр *Access* является комбинацией перечисленных ниже значений-флагов:

F_OK (**0**)	Проверка существования.
X_OK (**1**)	Проверка доступности выполнения.
W_OK (**2**)	Проверка доступности записи.
R_OK (**4**)	Проверка доступности чтения.

Параметр *Access* может быть опущен: его значением по умолчанию является 0. В этом случае, проверяется только само существование файла. Если же он отличен от 0, он рассматривается как комбинация значений-флагов, перечисленных выше. Результатом выполнения **f_access** является 0 (если проверка успешна), и -1 (если она неудачна). (Т.е. заметим, что **f_access** не предикат!)
Каталоги обычно тоже являются файлами, и имеют аналогичные файлам права доступа. «Выполнение» (*X_OK*) для них обычно означает право поиска файла в каталоге.

Изменение режима доступа к файлу

(Операция в основном аналогична системному вызову **chmod**(2) в UNIX.)

Изменение режима доступа к файлу	
Нормальная форма	***Семантика***
`f_chmod``(FileName, Mode)`	*Изменяет режим доступа к файлу FileName на Mode.*

Далеко не всегда режим доступа к файлу легко изменить. (Если вы не являетесь администратором — обычно, вы можете менять только режим доступа к файлам, владельцем которых вы являетесь.)
Параметр *Mode* является комбинацией следующих значений (восьмеричных!):

S_IXOTH (**1**)	Доступен для выполнения (всем пользователям)
S_IWOTH (**2**)	Доступен для записи (всем пользователям)
S_IROTH (**4**)	Доступен для чтения (всем пользователям)
S_IXGRP (\o**10**)	Доступен для выполнения (группе)
S_IWGRP (\o**20**)	Доступен для записи (группе)
S_IRGRP (\o**40**)	Доступен для чтения (группе)
S_IXUSR (\o**100**)	Доступен для выполнения (владельцу)
S_IWUSR (\o**200**)	Доступен для записи (владельцу)
S_IRUSR (\o**400**)	Доступен для чтения (владельцу)

Б**о**льшая часть этих значений специфична для UNIX-систем (и менее осмысленна для Windows).
Результатом функтора является 0 (если операция успешна) или -1 (если изменить режим доступа к файлу не удалось).

Переименование файла

(Операция в основном аналогична системному вызову **rename**(2) в UNIX.)

Переименование или перемещение файла	
Нормальная форма	***Семантика***
`f_rename``(FileFrom, FileTo)`	*Изменяет имя файла с FileFrom на FileTo.*

Операция переименовывает файл *FileFrom* в *FileTo*. (Переименование файла может одновременно сопровождаться его перемещением в другой каталог, если старое и новое имя относятся к разным каталогам.)
Даже если указанный файл существует — операция может кончиться неудачно (например, если доступ к нему для вас закрыт).
Результатом функтора является 0 (если операция успешна) или ненулевой *код ошибки* (если переименовать/переместить файл не удалось).

Удаление файла

(Операция в основном аналогична системному вызову **unlink**(2) в UNIX.)

Удаление файла	
Нормальная форма	***Семантика***
`f_remove``(FileName)`	*Удалить файл FileName.*

Операция удаляет файл *FileName* из файловой системы. Даже если указанный файл существует, операция может кончиться неудачно (например, если доступ к нему для вас закрыт). В большинстве UNIX-систем физическое удаление файла произойдет только тогда, когда он будет закрыт последним использующим его процессом.
Результатом функтора является 0 (если операция успешна) или ненулевой код ошибки (если

удалить файл не удалось).

Операции с каталогами

В большинстве операционных систем каталоги являются некой специальной разновидностью файлов. Возможность непосредственно читать и записывать их обычно недоступна. Обычно в процессе выполнения программы для нее определен *текущий каталог*.

Текущий каталог процесса	
Нормальная форма	***Семантика***
`get_curdir``()`	*Возвращает текущий каталог.*

Операция возвращает полное имя текущего каталога выполняющегося процесса (в виде строки). Для того, чтобы изменить его, используется следующая операция (аналог системного вызова **chdir**(2) в UNIX):

Изменить текущий каталог процесса	
Нормальная форма	***Семантика***
`chdir``(DirName)`	*Изменить текущий каталог на DirName.*

Имя нового каталога (*DirName*) может быть как абсолютным, так и относительным.
Для *создания* нового каталога используется следующая операция (аналог системного вызова **mkdir**(2) в UNIX):

Создать новый каталог	
Нормальная форма	***Семантика***
`mkdir``(DirName)`	*Создать каталог DirName.*

Операция возвращает 0 (при успехе), или ненулевой код ошибки (при неудаче). Созданный ею каталог является пустым, исключая специальные записи, такие, как «.» (для самого каталога) и «..» (для каталога уровнем выше).
Для *удаления* уже существующего каталога используется следующая операция (аналог системного вызова **rmdir**(2) в UNIX):

Удалить каталог	
Нормальная форма	***Семантика***
`rmdir``(DirName)`	*Удалить каталог DirName.*

Удаляемый каталог должен быть пустым (за исключением специальных записей «.» и «..»).
При успехе возвращается 0, при неудаче — код ошибки.
Для того, чтобы узнать полный список файлов, содержащихся в каталоге, проще всего использовать следующий итератор:

Итератор по каталогу	
Нормальная форма	***Семантика***
dir_loop(DirName, FileName, Body)	*Перебор файлов в каталоге DirName.*

Эта операция работает, как и все итераторы, перебирает все файлы, находящиеся в каталоге *DirName*. (Для текущего каталога, это должно быть ".") Имя каждого присваивается мутабельному аргументу *FileName* (как строка), после чего выполняется тело итератора *Body*. Таким образом, итератор выполняется столько раз, сколько файлов в каталоге *DirName* (и в соответствии с их физическим порядком).
Результатом выполнения является последний результат *Body* (или (), если *Body* не выполнялось ни разу).

Получение статусной информации о файле

Итератор **dir_loop** позволяет получить имена всех файлов, содержащихся в каталоге — но не более того. Для получения более детальных сведений файле (или каталоге), употребляется следующий функтор:

Информация о файле	
Нормальная форма	***Семантика***
f_stat(FileName)	*Получение информации о файле FileName.*

При успехе, результатом выполнения **f_stat** является список из 11 элементов. Его элементы (перечислены по порядку) являются целыми числами, имеющими следующий смысл:

0	*Device*	Номер устройства для UNIX (для Windows – номер диска)
1	*Inode*	Номер файла на устройстве (для Windows – обычно 0)
2	*RDevice*	Номер специального устройства для UNIX (для Windows – номер диска)
3	*Mode*	Тип файла и режим доступа к нему (см. **f_chmod**)
4	*Links*	Число связей файла (для Windows обычно 1)
5	*UID*	Идентификатор владельца файла (для Windows обычно 0)
6	*GID*	Идентификатор группы файла (для Windows обычно 0)
7	*Size*	Размер файла в байтах
8	*ATime*	Время последнего доступа к файлу (**Epoch**)
9	*MTime*	Время последнего изменения файла (**Epoch**)
10	*CTime*	Время последнего изменения статуса файла (**Epoch**)

Поля *Device* и *Rdevice* имеют разный смысл для UNIX-систем и Windows. Для UNIX они содержат комбинацию типа и логического номера дискового устройства; для Windows — номер буквы диска (0 — **A:**, 1 — **B:**, 2 — **C:**, и т. д.) Поле *Inode* (индексный номер файла) не имеет смысла для Windows.
Размер (*Size*) обычно является осмысленным только для регулярных файлов (не для каталогов).
Поля *UID* и *GID* имеют смысл только для UNIX.
Поля *ATime*, *MTime* и *CTime* измеряют время в секундах UNIX-эпохи (т. е. в секундах с полуночи 1 января 1970 года).

Аргументы выполнения

Любому выполняемому AWL-модулю может быть передан список аргументов (обычно, в командной строке).

Список аргументов	
Нормальная форма	***Семантика***
`_arguments``()`	*Список аргументов модуля.*

Результатом является список аргументов, переданных модулю при запуске. Все элементы списка являются строками.

Переменные окружения	
Нормальная форма	***Семантика***
`_environ``()`	*Словарь переменных окружения.*

Результатом являются все переменные окружения в виде хэша (словаря), с именами переменных в качестве ключей, и значениями. И те, и другие являются строками.